AFIT/GCS/ENG/95D-05

OBJECTSIM 3.0: A SOFTWARE ARCHITECTURE FOR THE DEVELOPMENT OF PORTABLE VISUAL SIMULATION APPLICATIONS

THESIS

Shawn Michael Hannan
Captain, USAF

AFIT/GCS/ENG/95D-05

AFIT/GCS/ENG/95D-05

OBJECTSIM 3.0: A SOFTWARE ARCHITECTURE FOR THE DEVELOPMENT OF PORTABLE VISUAL SIMULATION APPLICATIONS

THESIS

Presented to the Faculty of the School of Engineering of the

Air Force Institute of Technology
Air University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Systems
with
Emphasis in Software Engineering

Shawn Michael Hannan
Captain, USAF

December, 1995

Acknowledgments

A thesis effort can at times be a frustrating experience. Thankfully, I had mentors to keep fueling the fire when it got too low, and friends to throw water on the fire when it got too high. I needed both. Among the mentors, of course, was my thesis advisor, Major Mark Kanko. I thank him for all of his time and energy, but most particularly for bolstering my confidence when it was lacking, and his invariably positive outlook on the thesis effort.

Other mentors included LtCol (ret.) Patricia Lawlis, the originator of the software architecture research group, and Dr. Thomas Hartrum, my first software engineering mentor at AFIT. Both were also members of my thesis committee. I thank Colonel Lawlis for her guidance and continued devotion to our efforts, despite her retirement from the Air Force and subsequent relocation to Arizona. And I appreciate Dr. Hartrum's willingness to continue to try to teach me things, even though I never seem to learn anything.

I also want to thank the friends who helped me maintain my sanity during the latter half of the AFIT experience: Chuck Beem, who taught me that no matter how hard you work, there's always someone who works harder; Vince Hibdon, who taught me that no matter how little you do, you could always do a little less; and Don Hill, who taught me that no matter how busy you are, you always have time to re-accomplish something you've done 1000 times before.

Shawn M. Hannan

Table of Contents

List of Figures

List of Tables

Abstract

A visual simulation software architecture is a reusable design for visual simulation applications. This thesis effort was the third stage in an ongoing refinement of such an architecture, named ObjectSim. The primary goals of this stage were to improve the architecture by eliminating its dependence on two platform-specific graphics libraries (named GL and Performer, from Silicon Graphics, Inc.), and to examine the potential for expanding the architecture to accomodate distributed simulations.

The effort resulted in a new version of the architecture which allows the development of visual simulation applications which take full advantage of the aforementioned libraries without calling those libraries directly. This capability substantially improves the potential portability of future applications.

ObjectSim also has other enhancements not found in its predecessors, but still does not accommodate distributed simulations. Insights into addressing the distributed simulation issue are, however, included in this thesis.

OBJECTSIM 3.0: A SOFTWARE ARCHITECTURE FOR THE DEVELOPMENT OF PORTABLE VISUAL SIMULATION APPLICATIONS

1. Introduction

1.1 Overview

This thesis effort was a continuation of previous research in the area of visual simulation software architectures. A *visual simulation software architecture* is a reusable design for visual simulation applications. An implementation of such a design in a particular programming language is termed an *application framework.* The motivation for developing an architecture and associated application frameworks is increased productivity for application programmers. By offering these programmers a basic design as a starting point, a software architecture allows the developers to avoid starting from scratch with each new simulation. This document details the latest stages of an ongoing refinement of a visual simulation software architecture named ObjectSim.

1.2 Background

Students in the Graphics Laboratory (Lab) at the Air Force Institute of Technology (AFIT) have been conducting research in the field of distributed visual simulation for several years. This research has led to the development of a number of visual simulation applications, most notably a virtual cockpit [Dia94], a space modeler [Van94], a

commander's situational battle bridge [Roh94], and a debriefing tool for the Air Force's Red Flag excercise [For94].

In 1992, upon recognizing the many commonalities in these applications, a senior faculty member (LtCol Patricia Lawlis) suggested a common structure could be designed as the foundation for the various programs. The end result of this suggestion was a software architecture named *ObjectSim*, developed by Capt Mark Snyder [Sny93]. ObjectSim was the first real step taken by the Lab to incorporate software reuse into the graphics research. The four applications mentioned above were developed in conjunction with the new architecture, and all four continue to be based around the ObjectSim framework for visual simulations. Table 1 lists the software development problems experienced in the Lab **before** ObjectSim, as well as the noted improvements attributed to the reusable software architecture.

Problem	**Improvement**
No one responsible for standard components	ObjectSim provides a toolbox of components designed to fit together
No one responsible for evaluating and integrating new outside code	ObjectSim developer performed analysis in this area
Existing components hard to understand	Students had to understand ObjectSim approach, but less detail than before
No standard library locations, CASE, or CM to support large developments	ObjectSim captures a lot into one library, but does not solve CM problem
No design methodology adopted	ObjectSim facilitated a reasonable object-oriented approach to simulation design
Students don't have time to become well grounded in languages or design methods	ObjectSim provides design, reduces the amount of code necessary for success
Simulation projects didn't have stable requirements	ObjectSim allowed quicker maintenance turnarounds; simulation design stable

Table 1. Development Improvements with ObjectSim [Sny93, 73]

An application framework based on ObjectSim, written in C++, was completed in the fall of 1993. LtCol Lawlis then initiated a research effort to follow up on the success of ObjectSim. The goals of this research, conducted by Capt Jordan Kayloe, were to improve the architecture of ObjectSim and simultaneously explore an application framework in the Ada programming language [Kay94]. The culmination of this second effort was an architecture called *Easy_Sim*, and an implementation of that architecture in Ada 95. Easy_Sim was deemed a success, though there were several acknowledged shortcomings [Kay94, 137].

1.3 Research Motivation

The motivation for this research stemmed from two important issues unresolved by the ObjectSim and Easy_Sim efforts. The first concerned participation in distributed simulations. A distributed simulation is the aggregate of any number of simulations which are interacting with each other via a network. The ObjectSim and Easy_Sim architectures do not include components to assist applications in participating in such simulations.

The second issue concerned the tight coupling between the ObjectSim and Easy_Sim application frameworks and the underlying graphics software used in the Lab. This coupling leads to platform-dependent simulation applications which are difficult to port to other systems.

An additional driving force behind this effort was a desire to demonstrate the utility of Ada 95 in the realm of visual simulation. Although Easy_Sim proved Ada 95 a legitimate

option for implementing visual simulations, the C++ language still enjoys immense popularity in the graphics community, including the AFIT Lab. Ada continues to wage an uphill battle to establish itself as an accepted implementation language in this community.

1.4 Scope of Research

There were two main goals of this research effort. The first was to improve upon the predecessor architectures, ObjectSim and Easy_Sim. The second was to continue to explore the Ada 95 programming language as a tool for visual simulations.

In particular, with respect to the first goal, one objective was to design a visual simulation software architecture which could yield platform-independent application frameworks. Applications developed using the earlier frameworks are highly reliant on platform-specific software libraries, and the intention was to ensure the next generation of frameworks allowed the development of more portable applications. Furthermore, since ever-increasing Department of Defense (DoD) interest in distributed simulations is anticipated, a second objective was that the architecture specifically support these kinds of applications.

1.5 Research Approach

Since this effort was essentially an attempt to improve upon the successes of two earlier architectures, the first step was to examine those two predecessors. Because Easy_Sim was already an improvement over the original ObjectSim architecture, analysis was predominantly focused on the newer design. The primary objective during the examination of Easy_Sim was to locate framework dependencies on platform-specific software.

A natural second step was to seek out commercial architectures for comparison. It seemed quite likely that vendors would be developing products similar to ObjectSim and Easy_Sim, and that these products would offer insights into how to meet the objectives stated in the previous section.

Thirdly, key faculty members in the Lab were consulted for input. These faculty members are "customers" of the architecture in the sense that they oversee the development of visual simulations at AFIT. Current and previous Lab developers were also contacted for suggestions.

Given an analysis of the predecessor architectures, an examination of comparable commercial products, and the recommendations of knowledgeable Lab personnel, the final step was to develop and execute a plan for delivery of a third-generation visual simulation software architecture.

For the sake of consistency, it was decided to return to the original ObjectSim designation for this and future efforts.

1.6 Research Environment

Graphics work in the AFIT Lab is conducted on a network of Silicon Graphics, Inc. (SGI) workstations. The most powerful machines in current use are two Onyx RealityEngines. One of these is running version 5.2 of the IRIX operating system, while the other has been upgraded to version 5.3. In the following two subsections, brief descriptions of some of the key Silicon Graphics software used in the Lab are given.

1.6.1 The IRIS Graphics Library

The IRIS Graphics Library (GL) is a collection of low-level graphics routines which can be incorporated into graphics applications written on SGI platforms. The library provides simple capabilities such as drawing points, lines and polygons on-screen. It also assists with basic animation via double-buffering. Additional GL features include:

- handling of user input
- coordinate transformations
- hidden surface removal
- lighting effects
- atmospheric effects
- antialiasing
- surface textures

In short, GL is an excellent package for drawing, or *rendering*, pictures on-screen. The added features of double-buffered animation and user input handling allow GL to be used for such things as simple video games [McL92].

Although it is possible to use GL for the development of complex visual simulations, newer libraries have been developed which are better suited to these types of applications,

because they allow developers to work at a higher level of abstraction. The next subsection provides an introduction to one such library.

1.6.2 The IRIS Performer Library

In contrast to GL, the IRIS Performer Library was specifically designed to support the development of full-scale visual simulations [Har94, xix]. By using Performer, simulation developers can quickly construct entire scenes from previously-created models of terrain, buildings, aircraft, etc. Performer offers a variety of routines, based on database formats commonly used in the graphics community, to read in such models [Har94, 30].
Once a developer has used Performer to construct a visual scene, he or she can subsequently use Performer to manage the simulated motion within that scene. With simple calls to the Performer library, the developer easily adjusts the positions of entities and viewpoints within the scene. Performer then handles rendering the changes on-screen [Har94, 47].

Because simulations can be quite intensive in terms of computation, SGI has designed special hardware to support them. In particular, the previously-mentioned Onyx RealityEngines are highly-capable simulation workstations. The IRIS Performer library was developed with these workstations in mind, and the combination of special-purpose hardware and highly-tuned software leads to very effective simulations [Sch94, 1-2]. The drawback is that simulations written using SGI hardware and Performer software are not easily moved to other simulation platforms. There is also concern that reliance on specific

versions of hardware and software will impede efforts to transition the software to newer and better platforms when they become available, even if those platforms come from SGI.

1.7 Document Overview

The remainder of this document is divided into five chapters. Chapter 2, "Summary of Current Knowledge," covers a variety of information which is important background for readers of this thesis. Chapter 3, "The Development of ObjectSim 3.0," details the design of the latest architecture, including the routes which were considered but not chosen, and concludes with a discussion of ObjectSim's new design. Chapter 4, "Specification and Implementation in Ada 95," describes the translation of the ObjectSim architecture into an Ada application framework. Chapter 5, "Results and Comparisons," analyzes ObjectSim 3.0, and examines the advantages and disadvantages of the new architecture versus its predecessor. Finally, Chapter 6, "Recommendations for Future Study," lists the lessons learned during the course of this effort, as well as suggestions for continued work in this area.

2. Summary of Current Knowledge

2.1 Overview

The fundamental background information covered in this chapter is divided into six sections. First, in section 2.2, is a discussion of object-oriented topics, including introductions to Rumbaugh's Object Modeling Notation and the Ada 95 programming language. Next, section 2.3 briefly describes the features of three standard graphics libraries. Section 2.4 introduces the Distributed Interactive Simulation protocol for distributed visual simulations, and section 2.5 offers a short summary of Garlan and Shaw's "An Introduction to Software Architecture [Gar93]." Lastly, the final two sections of chapter 2 describe the ObjectSim and Easy_Sim architectures developed at AFIT.

2.2 Object-Oriented Concepts

Both ObjectSim and Easy_Sim were designed using object-oriented concepts. For an introduction to these concepts, see [Boo91] or [Rum91]. The next two subsections provide an overview of the object-oriented analysis and design notation generally used at AFIT, as well as an introduction to object-oriented programming with Ada 95.

2.2.1 Rumbaugh's Object Modeling Notation

Rumbaugh [Rum91] details techniques for analyzing, designing and implementing software systems in an object-oriented fashion. The intention of this subsection is not to summarize those techniques, but rather to familiarize the reader with the Rumbaugh notation, which will appear elsewhere in this document.

2.2.1.1 Classes

Of primary interest is Rumbaugh's notation for capturing the organization of, and relationships between, classes in a software system. Classes are drawn as simple rectangles, with the name of the class centered at the top of the rectangle. Attributes of the class may be listed below the name, and operations, or methods, may be listed below the attributes. A simple example is:

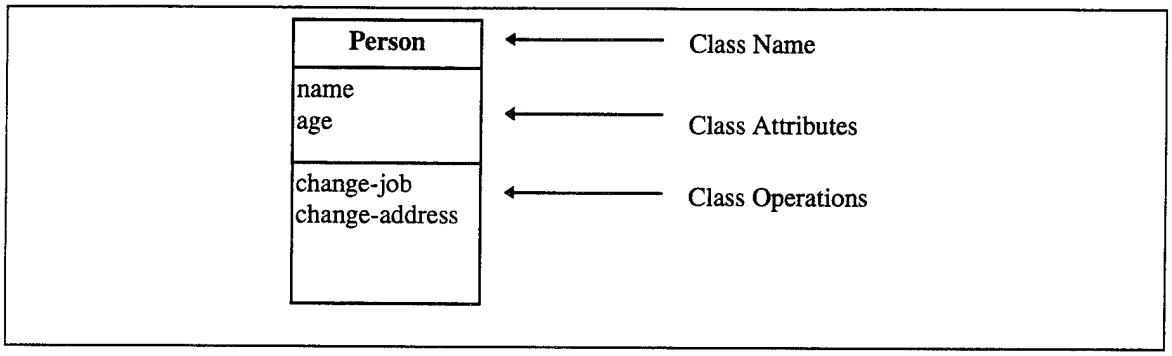

Figure 1. Person class [Rum91, 26]

2.2.1.2 Associations

Associations between classes are drawn as lines connecting the rectangles. The lines are usually labelled with identifiers which give meaning to the associations. For example, the following diagram asserts that objects of type **Person** work for objects of type **Company**. (The direction of associations is implied by the context [Rum91, 27]. In this case, the association reads from left to right.) Note that class attributes and operations have been omitted for simplicity.

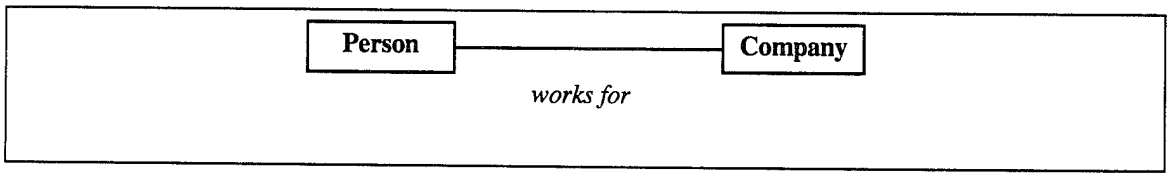

Figure 2. *Works for* association [Rum91, 34]

As it stands, the simple Rumbaugh diagram of Figure 2 shows a one-to-one relationship between **Person** and **Company**. The implication is that a person can only work for one company, and a company can only employ one person. Rumbaugh uses *multiplicity balls* to allow for multiple associations between classes. A solid ball means zero or more, and a hollow ball means zero or one [Rum91, 30]. For example, the following variation of Figure 2 shows how we would indicate that a company may employ zero or more people, and that every person may or may not work for a company. (This diagram does not allow for the possibility of a person working for more than one company.)

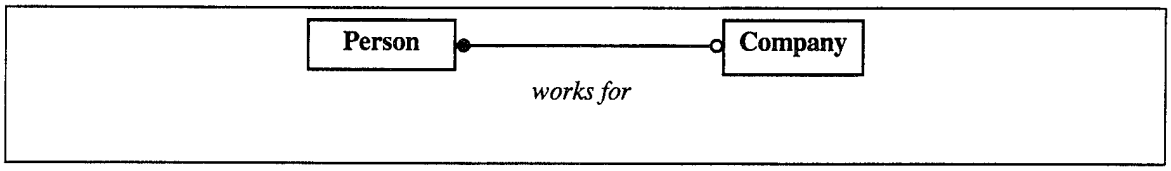

Figure 3. Adjusted *works for* association

2.2.1.3 Aggregation and Generalization

Two important kinds of relationships between classes are *aggregation* and *generalization.* Rumbaugh defines aggregation as

> the "part-whole" or "a-part-of" relationship in which objects representing the *components* of something are associated with an object representing the entire *assembly* [Rum91, 36].

Rumbaugh uses a diamond to capture this type of association. An example might be the relationship between a computer and its consituent parts:

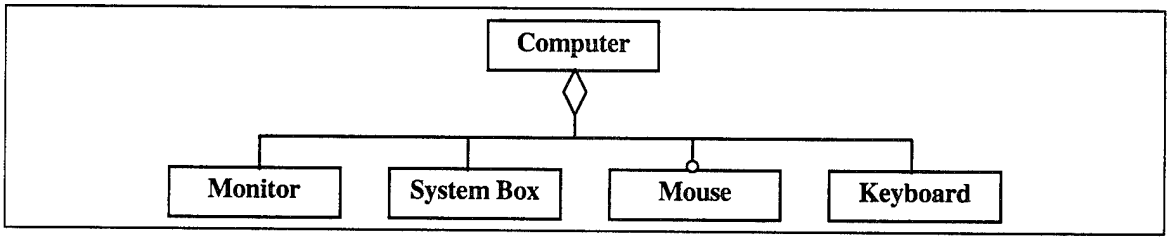

Figure 4. Aggregation [Rum91, 38]

Generalization, according to Rumbaugh, is a relationship between a class and one or more refined versions of that class. The original class is called the *superclass*, and the refined classes are known as *subclasses*. In a generalization association, subclasses inherit attributes and operations from their superclasses. This process is termed *inheritance* [Rum91, 39]. The following example of generalization shows how different types of employees inherit characteristics from an overall Employee superclass:

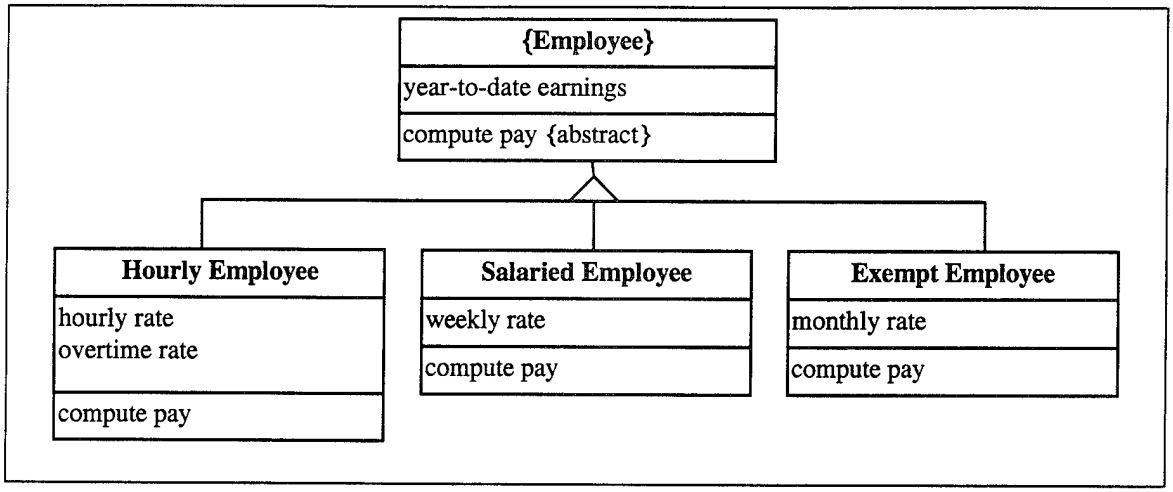

Figure 5. Generalization [Rum91, 62]

In Figure 5, the triangle beneath the **Employee** class indicates that the other three classes are subclasses of **Employee**. These three classes therefore inherit the **year-to-date earnings** attribute and the **compute pay** operation.

The braces around the word "Employee" specify that **Employee** is an abstract class, which means there will be no actual instances of the **Employee** class. All object instances will be required to be designated as either **Hourly Employee**, **Salaried Employee** or **Exempt**

Employee. (Note: Rumbaugh does not use braces around the class name to identify it as abstract. This is an extension invented for illustrative purposes in this thesis.)

Also, the designer has designated **compute pay** as an abstract operation, so as to require every subclass of **Employee** to include an algorithm for computing pay for that subclass. For this reason, the **compute pay** operation is shown in every subclass. The operations are implicitly different in each case. This in contrast to the **year-to-date earnings** attribute, which is also inherited by all subclasses, but only shown in the superclass.

2.2.1.4 Summary

The following example summarizes the Rumbaugh object diagram notation. In this figure, we see a description of the relationships among kinds of desk lamps. The diagram asserts a lamp is composed of a base, a cover, a switch, and some wiring. Inheritance is used to allow for two subclasses of **Lamp--Fluorescent Lamp** and **Incandescent Lamp**. The two subclasses include special components beyond the standard four mentioned previously.

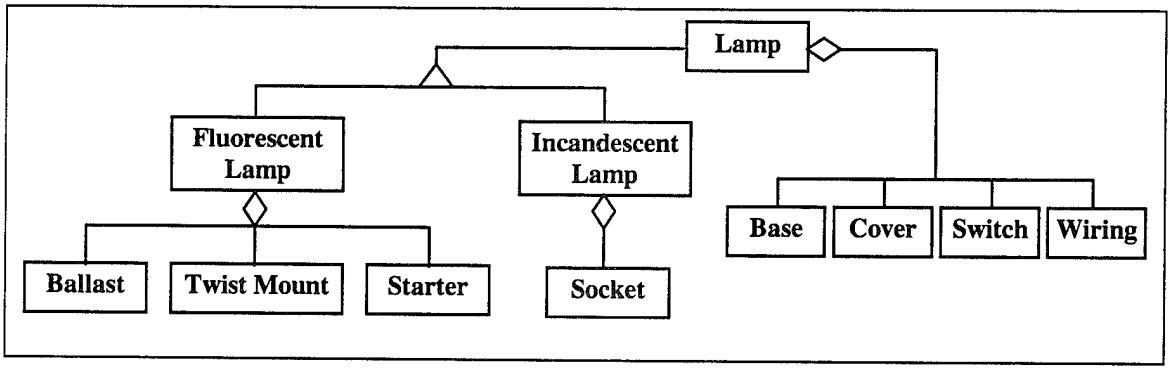

Figure 6. Aggregation and Generalization [Rum91, 59]

2.2.2 Object-Oriented Programming with Ada 95

Ada 95 is the first internationally-standardized object-oriented programming language [Ada95, 1]. This standardization implies a high degree of code portability amongst platforms. Ada applications should be easy to move from one computer to another, provided both computers have compilers which conform to the standards established by the Ada Reference Manual [RM95], and assuming the applications rely only on standard features.

For an in-depth discussion of Ada 95 the reader is referred to [RM95]. The intention of this section is only to show examples of what object-oriented programming in Ada 95 looks like. The examples are original, though the style is influenced by the conventions established in [Cer93].

Let us assume we wish to implement the following object-oriented design using Ada 95:

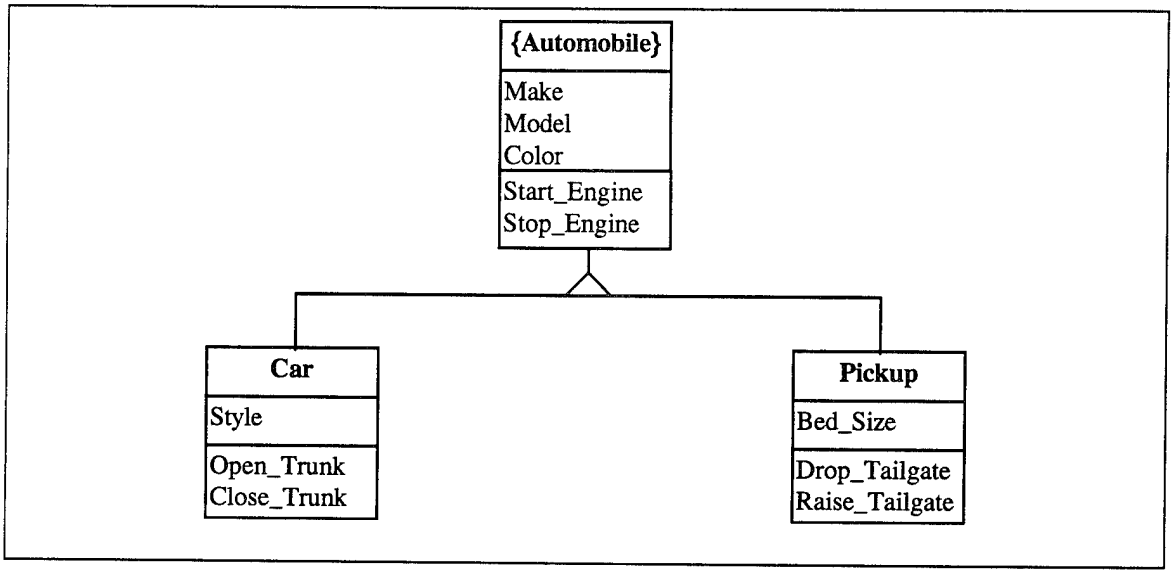

Figure 7. Automobile class hierarchy

Figure 7 shows a simple class hierarchy for automobiles. Our Ada 95 implementation of this hierarchy includes several packages. The first corresponds to the **Automobile** superclass, and is shown in Figure 8.

```
with Ada.Finalization;
package Automobile is

   -- This is the actual "class type."  It is always called "Object."
   -- Users of the package will declare variables such as
   --       My_Auto : Automobile.Object;

   type Object is abstract new Ada.Finalization.Controlled with private;

   -- The following type makes it easy for users to have pointers
   -- to object instances.
   type Reference is access all Object'Class;

   type Auto_Make is (Ford, Chrysler, General_Motors);

   type Auto_Model is (F_150, Concorde, Camaro);

   type Auto_Color is (Red, Green, Black);

   -- There are usually subprograms for changing and retrieving the
   -- attribute values.
   function Make_Of  (Instance : Object) return Auto_Make;

   function Model_Of (Instance : Object) return Auto_Model;

   function Color_Of (Instance : Object) return Auto_Color;

   procedure Set_Make  (Instance : in out Object;
                        Make     : in     Auto_Make);

   procedure Set_Model (Instance : in out Object;
                        Model    : in     Auto_Model);

   procedure Set_Color (Instance : in out Object;
                        Color    : in     Auto_Color);

   -- Here are the desired operations/methods.
   procedure Start_Engine (Instance : in out Object);
   procedure Stop_Engine  (Instance : in out Object);

private
   type Object is new Ada.Finalization.Controlled with
      record
         Make  : Auto_Make;
         Model : Auto_Model;
         Color : Auto_Color;
      end record;
end Automobile;
```

Figure 8. Automobile class in Ada 95

The two subclasses, **Car** and **Pickup**, are coded as *child packages* of **Automobile**. The class types are based on the abstract **Automobile** class type, and **Start_Engine** and **Stop_Engine** are inherited (as well as the **Make**, **Model**, and **Color** attributes). The **Car** package is shown in Figure 9. **Pickup** would be similar.

```
package Automobile.Car is

   -- This type definition allows us to inherit all attributes
   -- and primitive operations of type Automobile.Object.

   type Object is new Automobile.Object with private;

   type Reference is access all Object'Class;

   type Car_Style is (Sedan, Coupe);

   function Style_Of (Instance : Object) return Car_Style;

   procedure Set_Style (Instance : in out Object;
                        Style    : in     Car_Style);

   procedure Open_Trunk (Instance : in out Object);

   procedure Close_Trunk (Instance : in out Object);

private

   type Object is new Automobile.Object with
      record
         Style : Car_Style;
      end record;

end Automobile.Car;
```

Figure 9. Car subclass of Automobile in Ada 95

Finally, Figure 10 shows a short procedure which demonstrates how users might declare and manipulate an instance of the **Car** class.

```
with Automobile.Car;
procedure Test_Engine is

   Sports_Car : Automobile.Car.Object;

begin

   Automobile.Car.Set_Make  (Instance => Sports_Car,
                             Make     => Automobile.General_Motors);

   Automobile.Car.Set_Model (Instance => Sports_Car,
                             Model    => Automobile.Camaro);

   Automobile.Car.Set_Color (Instance => Sports_Car,
                             Color    => Automobile.Red);

   Automobile.Car.Set_Style (Instance => Sports_Car,
                             Style    => Automobile.Car.Coupe);

   Automobile.Car.Start_Engine (Sports_Car);

   Automobile.Car.Stop_Engine (Sports_Car);

end Test_Engine;
```

Figure 10. Test_Engine procedure manipulates a Car object.

2.3 Standard Graphics Libraries

In the search for platform-independence, the possible role of standard graphics libraries was considered. The following subsections briefly introduce three common graphics libraries which are available on a variety of platforms.

2.3.1 PHIGS

The Programmer's Hierarchical Interactive Graphics System (PHIGS) is a fairly recent international standard in the world of computer graphics. It gives programmers the capability to define and display two-dimensional and three-dimensional models with which users can interact. Because it is available for a wide range of hardware and operating system combinations, it is possible to write PHIGS applications which are highly portable between platforms [How91, 1].

PHIGS offers routines which support the following features:

- Graphical output primitives: lines, text and filled areas
- Attributes: such as the style of lines--dashed, dotted or solid
- Model Creation: combining output primitives with their attributes to form *structures*, and composing complex graphical models from these structures
- Model Display: actually displaying the previously-created models
- Model Editing: changing previously-created models
- Transformations and Viewing: moving models within a scene, and viewing the scene from various vantage points
- Input: reading user input to allow interaction with the graphical models
- Model and Picture Files: archiving model data

As the reader can see, the focus of PHIGS is on constructing and displaying graphical models [How91, 3]. The library does not include features to support real-time visual simulation.

2.3.2 OpenGL

The newest proposed standard graphics library is called *OpenGL*. Like Performer (discussed in chapter 1), OpenGL is an invention of Silicon Graphics. The crucial difference is that OpenGL is "an open standard designed to run on a variety of computers and a variety of operating systems" [Pro94,2]. Therefore, any software which relies on OpenGL should be easy to port to any platform which supports the OpenGL standard. Currently, OpenGL implementations exist for some Silicon Graphics workstations, as well as selected DEC workstations, IBM RS/6000 workstations, and all computers running Windows NT. Versions for other systems are under development.

According to Segal and Akeley, OpenGL "is very similar in both its functionality and its interface to Silicon Graphics' IRIS GL" [Seg93, 3]. The interface consists of several hundred subprograms which allow programmers to render high-quality graphical images, particularly of three-dimensional objects in color. As with IRIS GL, the focus is on rendering, not simulation. Unlike IRIS GL, however, OpenGL has no facilities for user input [Seg93, 4].

Although applications relying on OpenGL should, in general, be quite portable, it must be noted that OpenGL does not make applications entirely platform-independent. Certain crucial functions (such as opening a window on the screen) are not provided by OpenGL. Applications using OpenGL must therefore resort to platform-dependent operations in these cases [Pro94,2].

2.3.3 OpenInventor

OpenInventor is yet another product from Silicon Graphics, Inc. SGI describes this product as "an object-oriented 3D toolkit" for developing interactive graphics applications [Bel95, 1]. OpenInventor runs on top of OpenGL, which implies a high degree of portability of OpenInventor applications. Features include [Bel95, 1]:

- a standard file format for 3D data interchange
- a simple event model for 3D interaction
- animation objects called "engines"
- a programming model based on a 3D scene database
- a set of objects including cubes, polygons, cameras, lights and more

Counterbalancing the features of OpenInventor are, of course, certain disadvantages. These include a requirement to develop applications using C or C++, as well as limitations as to which compilers can be used [Bel95, 2]. Most important from the Lab's perspective is the fact that OpenInventor is not as well suited to visual simulation as SGI's Performer library. SGI says:

> ...Performer was designed for vis-sim, while Inventor was designed to be more general purpose. IRIS Performer is for developers who need to extract maximum performance from SGI machines for visual simulation, virtual reality, game development, and high-end CAD systems. Often these applications need multi-processor Onyx systems with multiple Reality-Engine pipelines with a high degree of parallelism and running at fixed frame rates.
>
> Inventor is designed for maximum programmer productivity when writing other kinds of 3D applications, like modelling, animation, visualization, etc. [Sch94, 3]

So although OpenInventor adds capabilities above and beyond the features of OpenGL, while still preserving a high degree of platform-independence for applications, it is not inherently well-suited to the development of high-performance visual simulations.

2.3.4 Summary

PHIGS and OpenGL are two contemporary graphics libraries which can be used to develop highly-portable rendering applications. Unfortunately, neither library was designed to support visual simulation. OpenInventor comes closer to this latter goal, but is still not comparable to SGI's Performer library. The result is a portability vs. performance dilemma for visual simulation developers. No library offers the power of Performer with the portability of OpenGL.

2.4 Distributed Interactive Simulations

The Distributed Interactive Simulation (DIS) protocol is a standard which allows geographically separated computers to participate in a joint simulation without the assistance of a central computer. The various simulation participants (tanks, airplanes, etc.) are termed *entities*. Entities interact by passing prescribed format messages known as *Protocol Data Units (PDUs)* [Bel93, 14].

Following are some of the key principles of the DIS protocol [She92, 14]:

- Each host maintains a local copy of information such as the terrain of the simulation and entity models.
- Each entity bases its viewpoint of the simulation on the information supplied by its host.
- Each host approximates the positions of other entities of interest so as to maintain the simulation without constant position updates from all entities.
- To minimize network data traffic, each host broadcasts updates only under certain conditions.

To expound on the last two principles outlined above, a definition of *dead reckoning* algorithms is required. Dead reckoning basically means approximating the new position of an entity based on its last known position and how it was moving. DIS supports a variety of algorithms for calculating such approximations. More accurate algorithms take longer to execute, and therefore slow down the simulation [She92, 20].

The DIS protocol requires that each host maintain **two** positions for local entities--the actual position and the dead reckoned position. The host must also maintain dead reckoned positions for all non-local entities. When a host recognizes the dead reckoned position of a local entity is inaccurate, it broadcasts the correct position to all other hosts.

The other hosts abandon their inaccurate approximations in favor of the new information. In general, this method leads to minimal network traffic [She92, 17]. The overall result is that each host is essentially running a local copy of the entire simulation.

2.5 Software Architectures

The term *software architecture* is fairly new and its definition continues to evolve. For our purposes, a software architecture can be defined as the high-level design (or perhaps *blueprint*) from which an application is constructed. This design is inherently reusable; many similar applications can be built using the same architecture. This reusability aspect is a primary motivation for developing software architectures, since reuse generally leads to resource savings during application development.

In one of the premier papers [Gar93] on the subject, Garlan and Shaw offer insights into the predominant styles of existing software architectures. They compare these styles in terms of *components* and *connectors*. Components are the computational building blocks used as the basis of an architecture. Connectors describe how the components interact with each other.

The first major type of architecture described by Garlan and Shaw is the *pipe and filter* style. In this kind of architecture, each component (filter) simply accepts data as input, performs some calculations, and provides output. The pipes provide the means by which input and output move amongst the filters. Each filter is independent of the others, and is thus interested only in providing the correct output based on its input. Most language compilers fall into this category, where the phases of compilation (lexical analysis, parsing, semantic analysis, code generation) can be viewed as stages in a pipeline.

A second type of architecture detailed by Garlan and Shaw is the *object-oriented* style. Components of this style are objects, and the connectors are subprogram invocations amongst them. The object-oriented style has become increasingly popular in recent years and is the style implemented for both ObjectSim and Easy_Sim.

In an *event-based, implicit invocation* architecture, components are modules whose interfaces provide both a collection of subprograms and a set of events. Components react to each other's events by executing subprograms. The connectors are thus characterized by the subprogram calls and the event announcements which cause them. Garlan and Shaw cite the Field System [Rei90] as an example. In this system, various tools register for a debugger's breakpoint events. When the debugger stops at a breakpoint, it announces an event. Other tools react to this event by automatically invoking the appropriate methods.

A fourth architectural style is the *layered system.* In this organization, the architecture consists of a hierarchy of layers and a set of protocols defining how the layers interact. Each layer is a component and each protocol is a connector. The best-known examples of this style are layered communication protocols, such as the Open Systems Interconnection (OSI) Reference Model [Tan88, 14].

The fifth type of architecture discussed by Garlan and Shaw is the *repository.* In this style, a central data structure holds the current state information for the system, and a collection of independent modules operate on this structure. The data structure and the modules collectively form the set of components. The connectors are somewhat ill-defined, as the authors state: "Interactions between the repository and its external components can vary significantly between systems [Gar93, 10]." Batch-sequential systems with global databases are examples of the repository architecture.

Beyond the architectural types outlined above, Garlan and Shaw also briefly discuss *domain-specific* software architectures, which describe organizational structures tailored to families of applications. An example is a visual simulation software architecture, which can be used as the foundation of any number of visual simulation systems. While ObjectSim and Easy_Sim are certainly examples of object-oriented architectures, both may be characterized as domain-specific as well.

2.6 ObjectSim

A Rumbaugh object diagram of the ObjectSim architecture is shown below. This diagram has been adapted from Figure 24 of [Sny93, 62] for clarity.

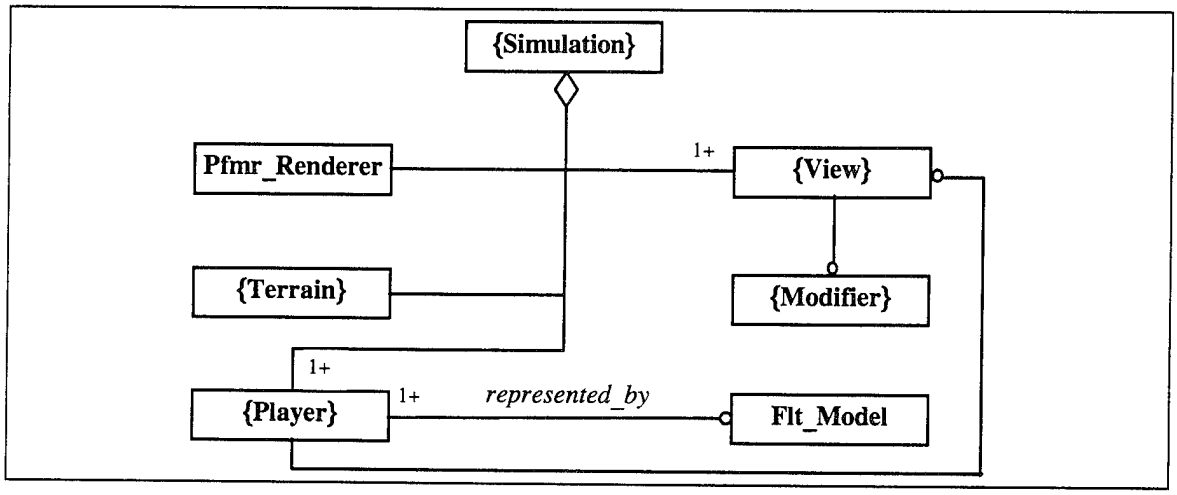

Figure 11. ObjectSim architecture (Note: "1+" indicates 1 or more)

The architecture shown in Figure 11 is used as the foundation of all ObjectSim simulations. The basic idea is that every simulation involves some sort of terrain, plus one or more moving entities (players), and one or more views into the simulation. Following are highlights of the ObjectSim architecture:

- The **Pfmr_Renderer** class was created in the hopes of isolating dependencies on SGI's Performer library. Unfortunately, this attempt at isolation failed, and Performer dependencies pervade the existing C++ application framework.

- Each view may be affected by input from the keyboard, a mouse, or other device. These devices come under the generic heading of *modifier*. Hence the **Modifier** class shown in Figure 11 [Sny93, 56].

- Most players in the simulation have an associated geometric representation. This representation is stored as an instance of the **Flt_Model** class [Sny93, 52], so named due to its reliance on a particular database format, the *Flight* format. Players may sometimes share the same representation. For example, two F-16 fighters might share a common **Flt_Model** instance.

- Each view must be attached to a player. As the player moves, so moves the view [Sny93, 48].

- A *stealth view* is a view which is attached to a player which has no associated geometric representation. The result is an invisible viewpoint which moves around the simulation [Sny93, 48].

2.6.1 ObjectSim and DIS

Throughout [Sny93], it is clear that allowing for participation in DIS simulations was a consideration during the design of ObjectSim. Although the ObjectSim architecture as presented in [Sny93] did not include classes for the development of DIS-participating simulations, the final C++ framework was modified to accomodate these applications [Sny93, 57-61]. Unfortunately, scrutiny reveals ObjectSim's DIS interface as a mystifying portion of the application framework. The networking class hierarchy is hard to unravel, as evidenced in Figure 12, adapted from Figures 23 and 25 of Snyder's thesis.

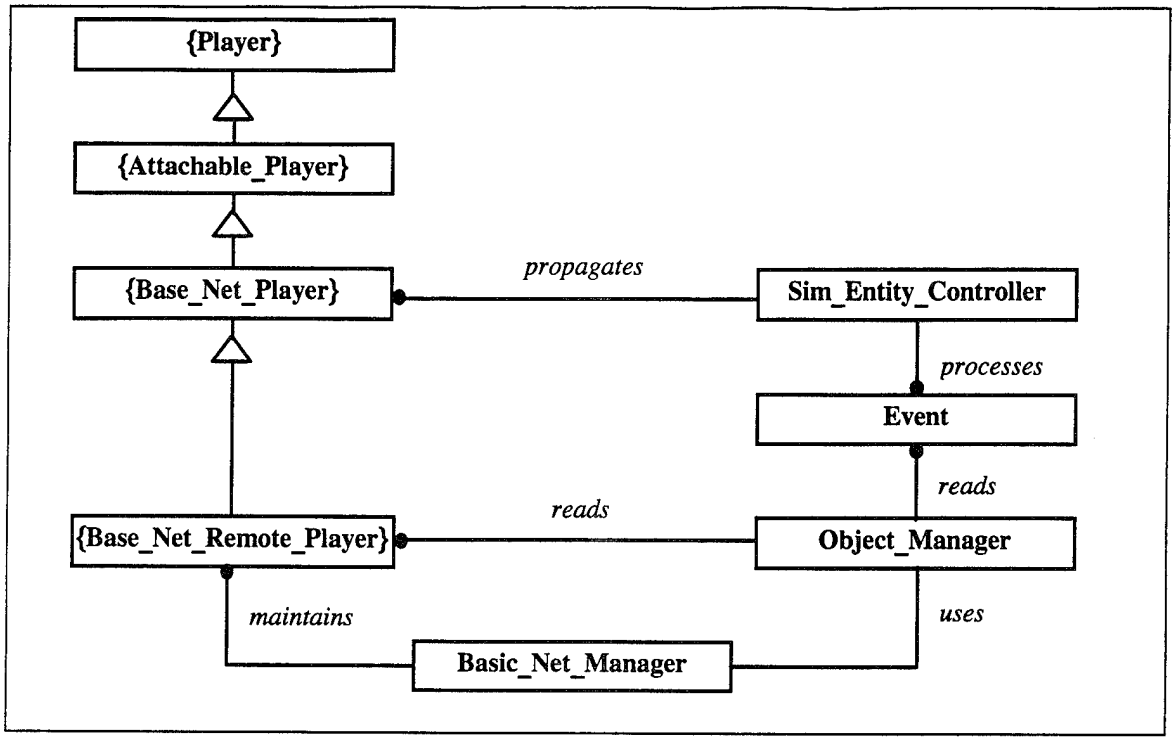

Figure 12. Networking classes in ObjectSim

The basic idea behind Figure 12 is as follows. The **Object_Manager** is the class which is actually responsible for interacting with the network. It retrieves two types of information--events and player updates. These two types of information are represented by the **Event** and **Base_Net_Remote_Player** classes, respectively. **The Basic_Net_Manager** is responsible for maintaining the list of remote players, and the **Sim_Entity_Controller** is responsible for propagating network events to local players.

It is interesting to point out that applications participating in DIS simulations are also required to **send** updates to the network. ObjectSim does not, however, arrange for this. ObjectSim applications must therefore assume full responsibility for keeping remote participants apprised of the local status.

In summary then, the ObjectSim architecture does not include components for distributed simulation. However, the associated C++ application framework does include classes for applications wishing to participate in DIS simulations. But those classes only concern **receiving** information from the network, which is only half of the problem. Furthermore, this portion of the framework is at best hard to follow, and at worst confounding.

2.6.2 ObjectSim and Platform-Independence

With simulation performance a top concern, ObjectSim was designed to use SGI's Performer library to its full potential. In his thesis, Snyder states:

> ObjectSim currently is dependent on the SGI platform and the Performer library. To make the architecture platform-independent would require a tree-based rendering abstraction similar to the Performer tree [Sny93, 99].

To put it simply, an effort to move ObjectSim to a new platform would require one of three things:

- Performer on the new platform
- something as close to Performer as possible on the new platform, and commensurate changes to the architecture
- a complete redesign and reimplementation of ObjectSim

2.7 Easy_Sim

ObjectSim was used as a starting point for the Easy_Sim effort of 1994. The end result was an architecture which was similar to, yet different from, ObjectSim. The Rumbaugh diagram shown in Figure 13 is an adaptation of Figure 23 from Kayloe's thesis.

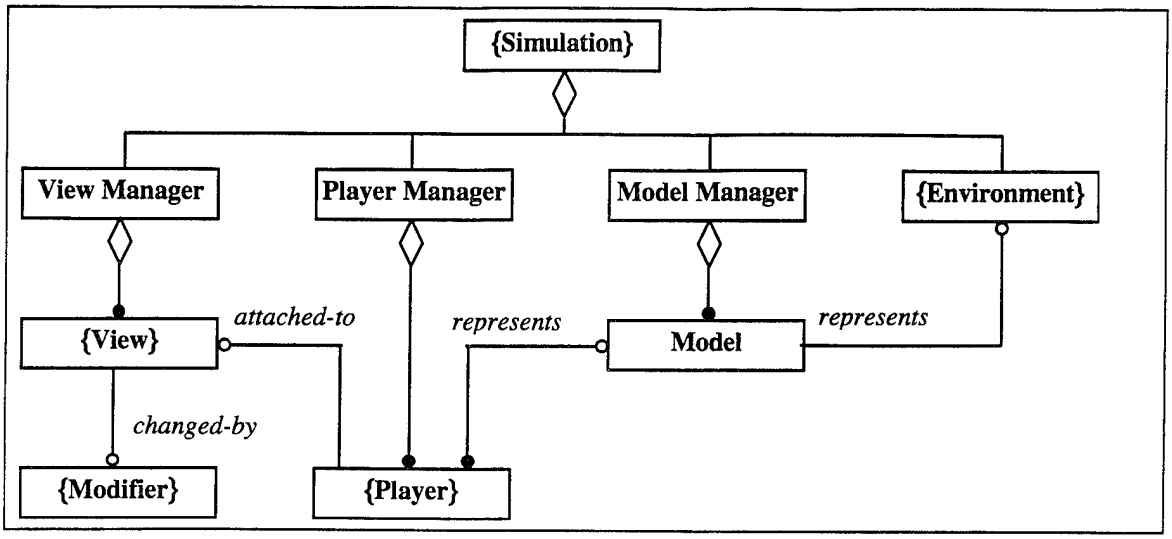

Figure 13. Easy_Sim architecture [Kay94, 79]

Like the ObjectSim architecture, Easy_Sim includes views, modifiers, players and models. Easy_Sim adds manager classes for players, views and models, and renames ObjectSim's **Terrain** class to **Environment**. The following subsections take a closer look at Easy_Sim's classes. The intention here is to hit the main points of how Easy_Sim works. For a more in-depth discussion, see [Kay94].

2.7.1 The Simulation Class

The **Simulation** class (shown in Figure 14) is essentially the aggregate of four other classes--the three manager classes and the **Environment** class. As Kayloe states in his thesis, the **Simulation** class is "the glue that holds the other pieces of the application together [Kay94, 76]." Thus, any simulation constructed using the Easy_Sim architecture will include one instance of each of the manager classes, as well as some sort of environment. The **Simulation** class is abstract, meaning developers must devise specialized subclasses of **Simulation** for actual use in applications. The only attribute of

this class is a **Pipe**, which is a Performer term for a rendering pipeline. Thus, one might argue that the reliance of the Easy_Sim application frameworks on the Performer library found its way "backwards" into the architecture itself.

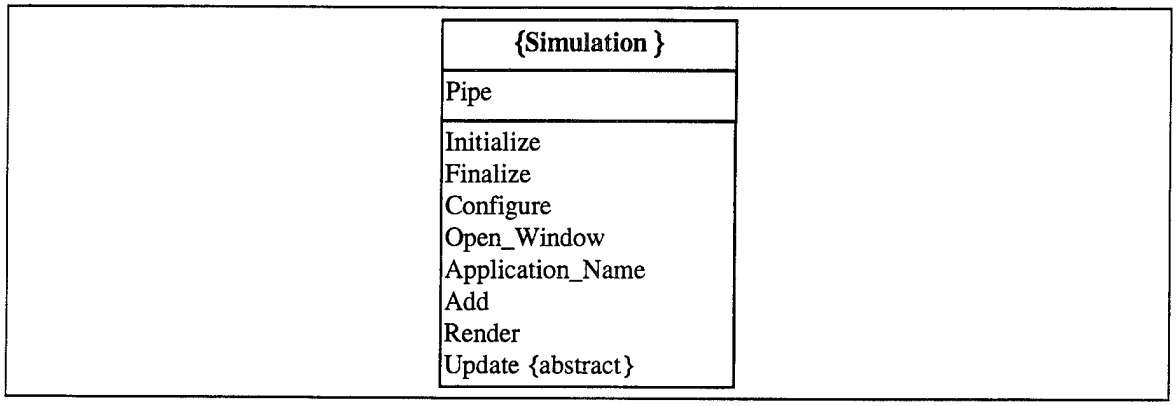

Figure 14. Easy_Sim's Simulation class [Kay94, 77]

2.7.2 The Player Class

Easy_Sim's **Player** class (depicted in Figure 15) is also abstract. When writing simulations, developers create subclasses of **Player** such as **Circling_Drone**, **Patrolling_Tank**, etc. In these subclasses, developers provide concrete **Update** operations which characterize the movement and other changes the players go through during the simulation. The **Update** operation for each player is called by the **Player_Manager** during each frame of the simulation.

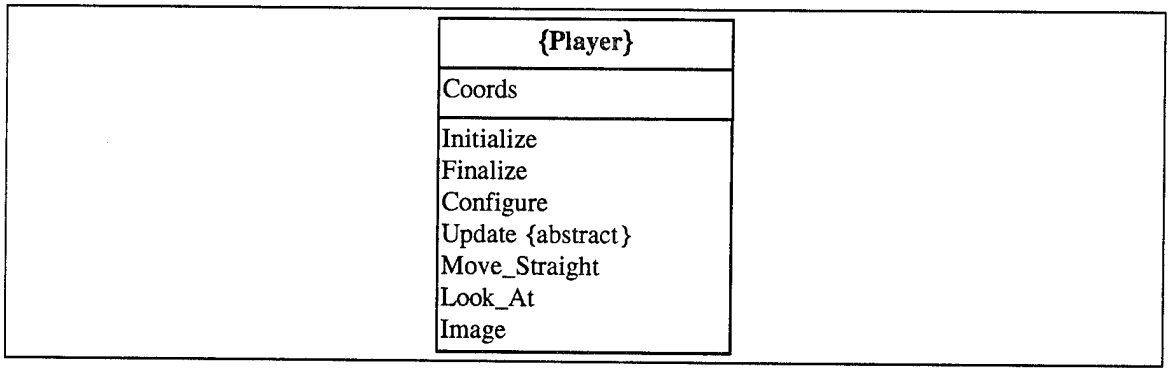

Figure 15. Easy_Sim's Player class [Kay94, 65]

2.7.3 The Model Class

Easy_Sim's **Model** class is fairly simplistic. Its purpose is to handle the geometric representations of simulation images. Predominantly, this means the class is used to read models from previously-created database files. Once loaded, models are associated with the environment and players via routines in other classes. As with the **Simulation** class, the only attribute of **Model** is an artifact of a framework reliance on Performer. (Abstractly, there is no such thing as the **root** of a model. The attribute only makes sense in a Performer context, where the root is used to attach the model to the scene database maintained by Performer.) The **Model** class is shown in Figure 16.

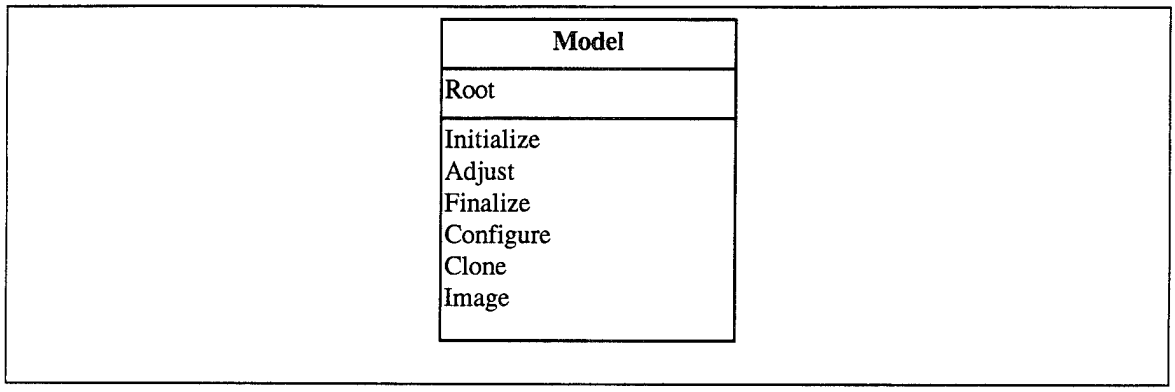

Figure 16. Easy_Sim's Model class [Kay94, 60]

2.7.4 The View Class

Easy_Sim's **View** class provides the capabilities for creating and manipulating a vantage point in a simulation. As Figure 17 shows, the **View** class has three attributes, two of which (**Channel** and **Scene**) are directly attributable to the frameworks' Performer dependencies. The third attribute is for the position and orientation of the view relative to the player to which it is attached. As with ObjectSim, every view must be attached to a player. As the player moves, the view moves.

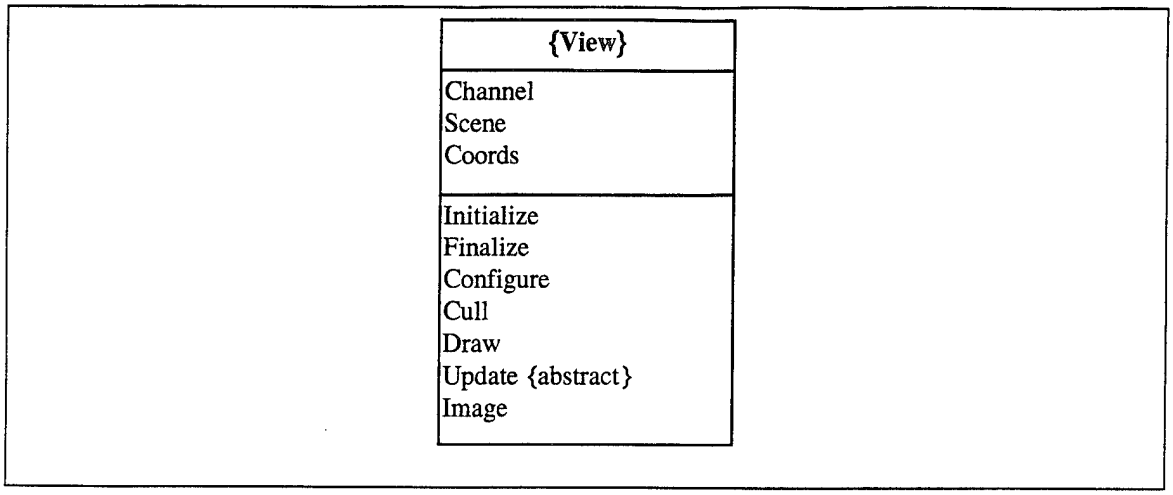

Figure 17. Easy_Sim's View class [Kay94, 69]

2.7.5 The Modifier Class

The **Modifier** class (Figure 18) is the third abstract class in Easy_Sim. Each subclass of **Modifier** corresponds to some sort of input device. When a developer designs a new subclass, he or she provides an **Update** operation to specify how input values are to be translated into position and orientation adjustments. One **Modifier** subclass is provided with Easy_Sim: **Standard_Input**, which accepts input from a mouse and keyboard.

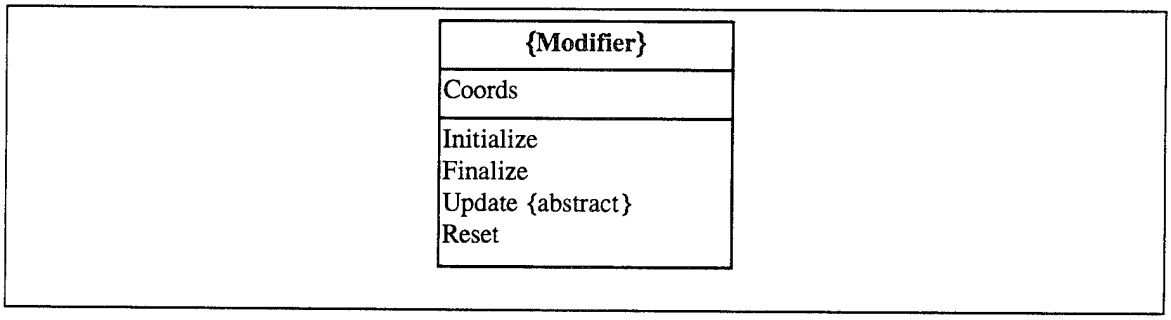

Figure 18. Easy_Sim's Modifier class [Kay94, 70]

2.7.6 The Environment Class

ObjectSim's **Terrain** class is replaced in Easy_Sim by an **Environment** class (Figure 19), which substantially differs from its ObjectSim counterpart. In ObjectSim, **Terrain** is simply a subclass of **Model**, though this isn't necessarily intuitive. Easy_Sim's approach seems to make more sense. **Environment** is an abstract class, and developers are free to design any number of subclasses, such as a heavenly spacescape, an undersea world, or a mountainous landscape. Easy_Sim provides one such subclass, called **Terrain**, which is a basic horizon and sun environment. **Terrain**, as well as every other kind of environment, has an associated **Model**, which contains the geometric representation to be drawn on-screen.

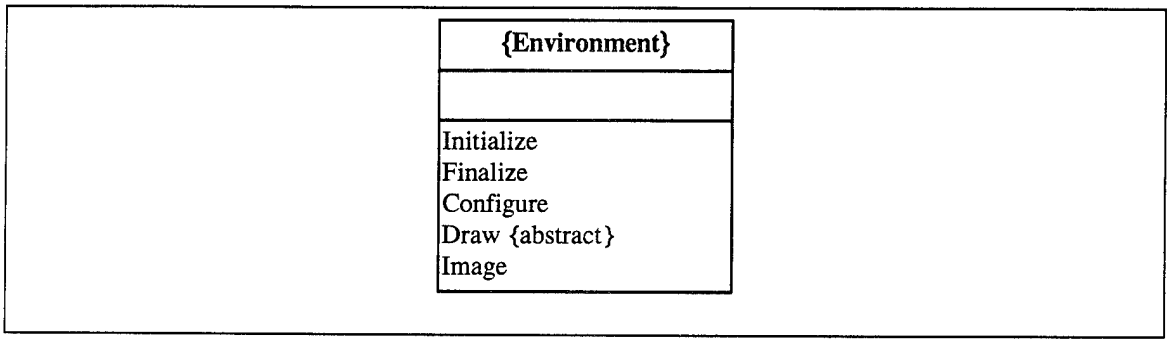

Figure 19. Easy_Sim's Environment class [Kay94, 62]

2.7.7 The Manager Classes

Easy_Sim's approach to dealing with multiple instances of the various classes is more consistent than that of ObjectSim. In Easy_Sim, the **Model**, **Player** and **View** classes each provide an abstraction of a single entity. Three additional classes, **Model_Manager**, **Player_Manager** and **View_Manager**, provide abstractions of **collections** of entities. **Model_Manager** and **Player_Manager** manage lists of models and players, respectively.

View_Manager maintains both a list of views and a list of view states [Kay94, 74].

Figure 20 shows the manager classes.

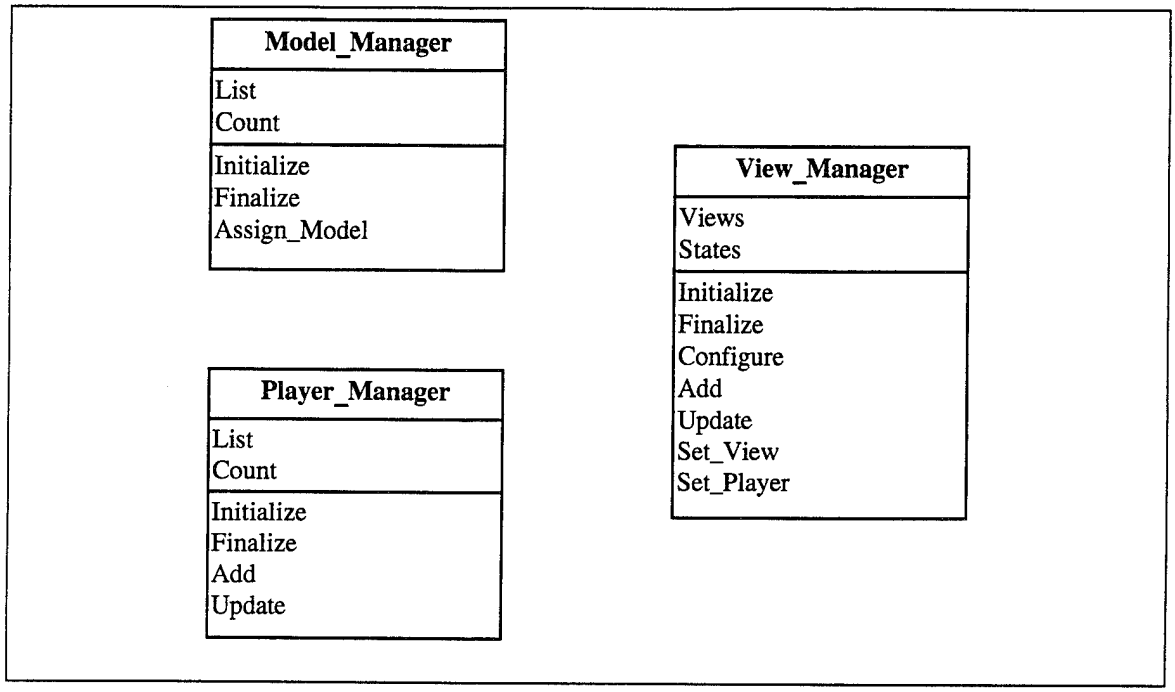

Figure 20. Easy_Sim's manager classes [Kay94, 72]

2.7.8 Easy_Sim and DIS

Easy_Sim includes no specific provisions for participation in DIS (or other distributed) simulations. In his thesis, Kayloe does make recommendations as to how Easy_Sim might be adjusted to accomodate distributed simulation. Specifically, he suggests the **Player_Manager** class might be replaced with an abstract class which allows for DIS participation. An appropriate subclass would be derived for each different simulation, based on that simulation's networking requirements [Kay94, 73]. To date, however, no simulation based on Easy_Sim has ever participated in a DIS simulation, using this approach or any other.

2.7.9 Easy_Sim and Platform-Independence

Since Easy_Sim served as the starting point for this effort, and a primary research goal was an architecture which would yield highly-portable application frameworks, it is important to understand what portions of Easy_Sim are **not** platform-independent. As a starting point, Figure 21 shows the layering of the Lab's visual simulation software. Higher level layers make calls into the lower level layers they touch.

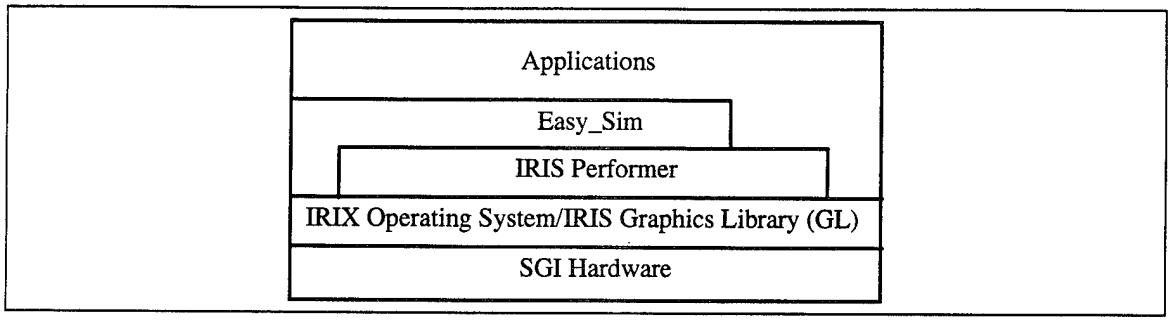

Figure 21. AFIT Lab software layering diagram

The software layering diagram shows how the IRIX Operating System and IRIS Graphics Library both interact directly with the SGI hardware, the Performer library is dependent on the lower-level software, and the Easy_Sim frameworks access both Performer and GL. Visual simulation applications have access to **all** layers, from Easy_Sim down to the operating system.

If the objective is to create applications which are relatively easily moved to other hardware, it is clearly important to restrict access to the specialized software which runs only on the SGI platforms. The implication is that applications should predominantly make calls to an abstract interface which hides access to this specialized software. This

leads to the crucial question: Does Easy_Sim provide an abstract interface which hides access to SGI's software?

In partial answer to that question, Kayloe states:

> Ideally, the Easy_Sim architecture should be portable, and its underlying layers should be interchangeable with a graphics library from any platform. There is nothing inherent in the Easy_Sim architectural design that prevents this adaptation, but no industry standards for graphics libraries currently exist. The architectural connectors that allow interaction between the Easy_Sim and Performer layers therefore operate differently than connectors to other commercial graphics libraries would operate [Kay94, 79-80].

Two points must be made concerning this statement. First, the portion of the statement concerning standard graphics libraries should be qualified. As discussed in chapter 2 of this thesis, there **are** industry standards for graphics libraries. The real point raised by Kayloe is that Easy_Sim needs the capabilities of Performer, and there are no standard graphics libraries which come close to providing these capabilities.

Second, there is a case to be made against the claim, "There is nothing inherent in the Easy_Sim architectural design that prevents this adaptation..." As pointed out earlier in this chapter, certain class attributes are pure artifacts of the Easy_Sim frameworks' reliance on Performer. If these attributes were removed at the design level and relegated to the role of implementation details within the frameworks, the architecture itself would have no fundamental dependency on the SGI software. Given this adjustment, the next step would be to ensure the application frameworks restrict access to SGI's specialized

software to the greatest extent possible. Consider the following excerpt from the Ada framework's specification of the **Player** class. Note the overt usage of Performer types.

```
with Easy_Sim.Model;
package Easy_Sim.Player is

   -- Instance is an instance of the Player class.

   procedure Configure (Instance   : in out Object;
                        Under_Node : in     Performer_Pf.PfNode);

   .
   .
   function Image (Instance : Object) return Performer_Pf.PfGroup;
   .
   .
end Easy_Sim.Player;
```

Figure 22. Easy_Sim framework's reliance on Performer

Figure 22 offers an example of how the Easy_Sim application frameworks allow platform-dependencies to find their way into applications, despite the platform-independent nature of the architecture itself.

The bottom line is that the Easy_Sim architecture, for the most part, is not fundamentally dependent on any platform. The key issue is ensuring the frameworks correctly implement the architectural design without allowing application reliance on platform-specific software.

3. The Development of ObjectSim 3.0

3.1 Overview

As discussed in section 1.2, one of the primary objectives behind this thesis was to define a platform-independent architecture which could be used as the foundation of all visual simulations at AFIT, including DIS-participating simulations. The initial analysis of previous work suggested three distinct avenues for the definition of this architecture. The next two sections outline the possibilities which were considered but rejected. The remainder of the chapter describes the third avenue, the design of ObjectSim 3.0, and the rationale behind that design.

3.2 Work with a Standard Library

The first approach to the development of a new architecture centered on existing standard graphics libraries. The libraries discussed in Chapter 2, for example, all offer certain capabilities on a variety of platforms. The question arose as to whether one of these libraries could be expanded to support visual simulation, or if perhaps a visual simulation architecture could be implemented using one of these libraries. The software layering diagram in Figure 23 depicts this latter scenario.

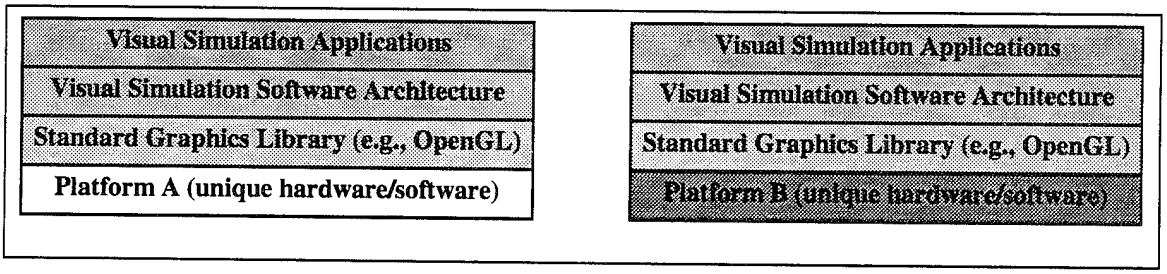

Figure 23. A possible approach: A visual simulation software architecture built on top of a standard graphics library

The approach shown in Figure 23 was attractive because it shifts the responsibility for platform-independence to the implementor of the standard graphics library. If a visual simulation software architecture could be constructed to rely only on, for example, OpenGL, the architecture and all applications which depend on it would port easily to any platform which supports the OpenGL library.

Unfortunately, the analyses of selected graphics libraries (discussed in chapter 2) led to concern about the apparent gap between features offered by standard libraries versus capability requirements of visual simulations. This is the same concern raised by Kayloe in his thesis (See section 2.7.9.).

Independently, Janett, Hayes and Miller reached similar conclusions:

> Interfaces such as OpenGL, Renderman and PHIGS+ provide extensive support for describing graphics primitives and for forming hierarchies of primitives, but are awkward for describing the composition of the virtual environment [Jan94, 4].

Janett et al. deemed the aforementioned libraries too low level for visual simulation, and also determined it would be difficult to extend or build upon them to "support the needs of the interactive simulation community [Jan94, 4]." They also concluded that any such extension would be overly complex [Jan94, 2].

As a result of these findings, the approach of designing an architecture around an existing standard graphics library was abandoned.

3.3 Adapt a Commercial Architecture

While the Lab has no intention of fostering new dependencies on specific commercial software, it seemed plausible that a commercial architecture might be examined and then adapted or somehow reimplemented in the Lab. Research along this path did not discover a suitable architecture, but two products were found which subsequently influenced the design of ObjectSim 3.0. The following subsections describe the two products and their pertinence to this thesis.

3.3.1 Clip

Loral's Advanced Distributed Simulation Division in Bellevue, Washington has been conducting work which might well contribute to the development of a platform-independent visual simulation software architecture. In the abstract of [Jan94], Janett *et al.* state:

> A standard Computer Image Generator (CIG) interface for real-time interactive visual applications that allowed software to be ported easily between many rendering platforms would provide many benefits to the VR, simulation and entertainment communities [Jan94, 1].

[Jan94] goes on to describe *Clip*, a proposed foundation for a standard real-time visual simulation interface. Clip was designed to allow the development of visual simulations which do not rely on specific graphics hardware and software. The authors echo the concern of the AFIT Lab when they say, "Without a standard interface, large sections of simulation code become tightly bound to a particular CIG [Jan94, 1]." (Note: Janett *et al.* use the term "Computer Image Generator", or CIG, in roughly the same context this thesis uses the term "platform.")

In the Clip Interface Design Document (IDD) [IDD94, 3-2], Loral lists the goals which influenced the design of Clip. The interface was to:

- be a logical and abstract, rather than physical and concrete, representation of the capabilities of the underlying CIG hardware and software.
- be tailored toward distributed simulation applications.
- be simple and uniform.
- allow full access to underlying CIG system capabilities.
- be portable across CIG vendors.
- be extensible.

Clip is just one part of a larger system known as *Vistaworks*, which is a collective title for the following software components:

- Real-Time Director - image generator graphics package which is responsible for rendering visual images
- S1000 - tools for creating models and terrains
  - a compiler which converts raw data into a form usable by the Real-Time Director
  - API (for accessing data from applications)
- SNIP - network interface package
- CLIP - image generator interface package
- TestSim - sample application
- Stealth - sample application

The first four components are used in combination to develop and run visual simulations such as the TestSim and Stealth applications provided as samples.

When a Vistaworks simulation is running, there are three independent processes active. These are:

- Simulation (Sim) - This is the process which is actually executing the specific application. It is responsible for handling user input and controlling the simulation via calls to the Clip basic functions. Examples include the Stealth and TestSim applications.

- Database and Polygon Processing (DPP) - Responsible for drawing the simulation on the Computer Image Generator (CIG). The way this works is entirely hidden from the application (and application developers), though Janett offered a few insights via email [Jan95a], and these are reflected in Figure 24.

- Clip-to-CIPE (CTC) - CIPE is the CIG Interface Processor Executive, which is a front-end to the Real-Time Director. The CTC process is responsible for translating Clip calls into commands to the CIPE.

There are actually two distinct parts to Clip. First, there's the interface through which applications manipulate the various aspects of the simulation. This interface is often referred to as the *application side* of Clip. The second part is known as the *CIG side* of Clip. This portion is embedded in the CTC process. During the course of a simulation, the two halves of Clip communicate with each other to accomplish the developer's objectives. The strong separation of the package into these two halves is what provides the desired platform-independence.

Figure 24 depicts a Vistaworks simulation. Other than handling user input and interacting with network entities (if there are any), the application controls all aspects of the visual simulation via calls to the Clip library. Clip views the entire simulation as a collection of interacting objects. The CIG, each individual display, each viewpoint, each model, and so on, are all instances of Clip classes. These classes are defined in the Clip IDD. (Unfortunately, the IDD is strongly influenced by the existing C implementation of Clip. The IDD is not so much an abstract definition of the Clip approach as a high-level commentary on the C implementation.)

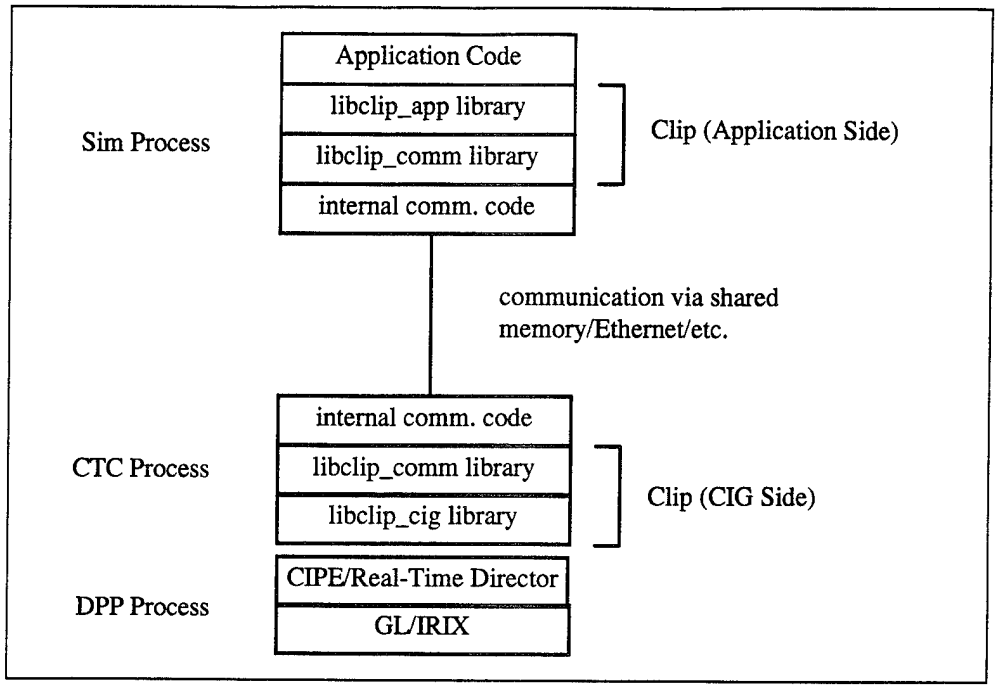

Figure 24. A Vistaworks simulation on an SGI platform

At run-time, the Clip classes are maintained within the CTC process. This means that many application requests for services require messages be sent from the Sim process to the CTC process, and the results of the requests often cause messages in the other direction. Furthermore, for the sake of speed, the application generally doesn't wait for the return message to arrive. Instead, the application continues processing. Eventually, Clip calls a *callback function* in the application to inform the application of the results of the earlier service request.

Following is an example of an application call to Clip:

```
status = clip_call(model_3d_table,                          /* object instance */
                   CLIP_GET_CANDIDATE_COMPONENT_INFO_BY_INDEX, /* method name */
                   &call_status,                              /* return status */
                   &ccinfo[i],                                  /* parameters */
                   _getting_ccinfo);                       /* callback function */
```

The above call results in a message being sent from the Sim process to the CTC process. During the message propagation time, the application continues executing. Eventually, the CTC process calls `_getting_ccinfo`, returning any desired data.

3.3.1.1 Clip Classes

Figure 25 shows the key Clip classes and their interrelationships. Instances of these classes appear in every Vistaworks simulation. The key classes are:

- **World** - an entire viewable scene, including the terrain, models of entities, weather conditions, etc.
- **CIG** - computer image generator; represents the device responsible for generating images of the simulation; (Note a physical monitor is not an example of a CIG. Rather, a workstation is an example, whereas a monitor is represented by an instance of the Display class.)
- **Display** - A CIG may have multiple attached displays. Each instance of the Display class corresponds to an output from a CIG.
- **View** - represents "what you see"; not just a viewpoint or an invisible camera, because it includes attributes such as sky color, horizon glow, and level of detail

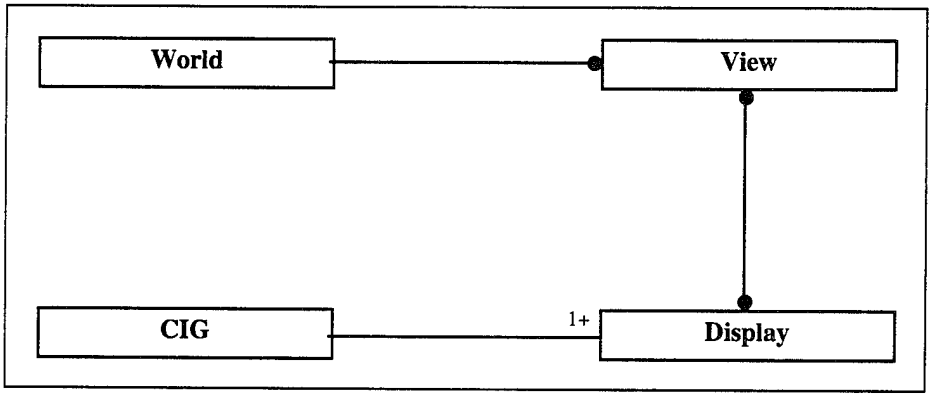

Figure 25. Key Clip classes and relationships in a Vistaworks simulation

In total, the Clip IDD describes 37 classes, some of which are yet to be implemented. These classes are hierarchically arranged as shown in Figure 26.

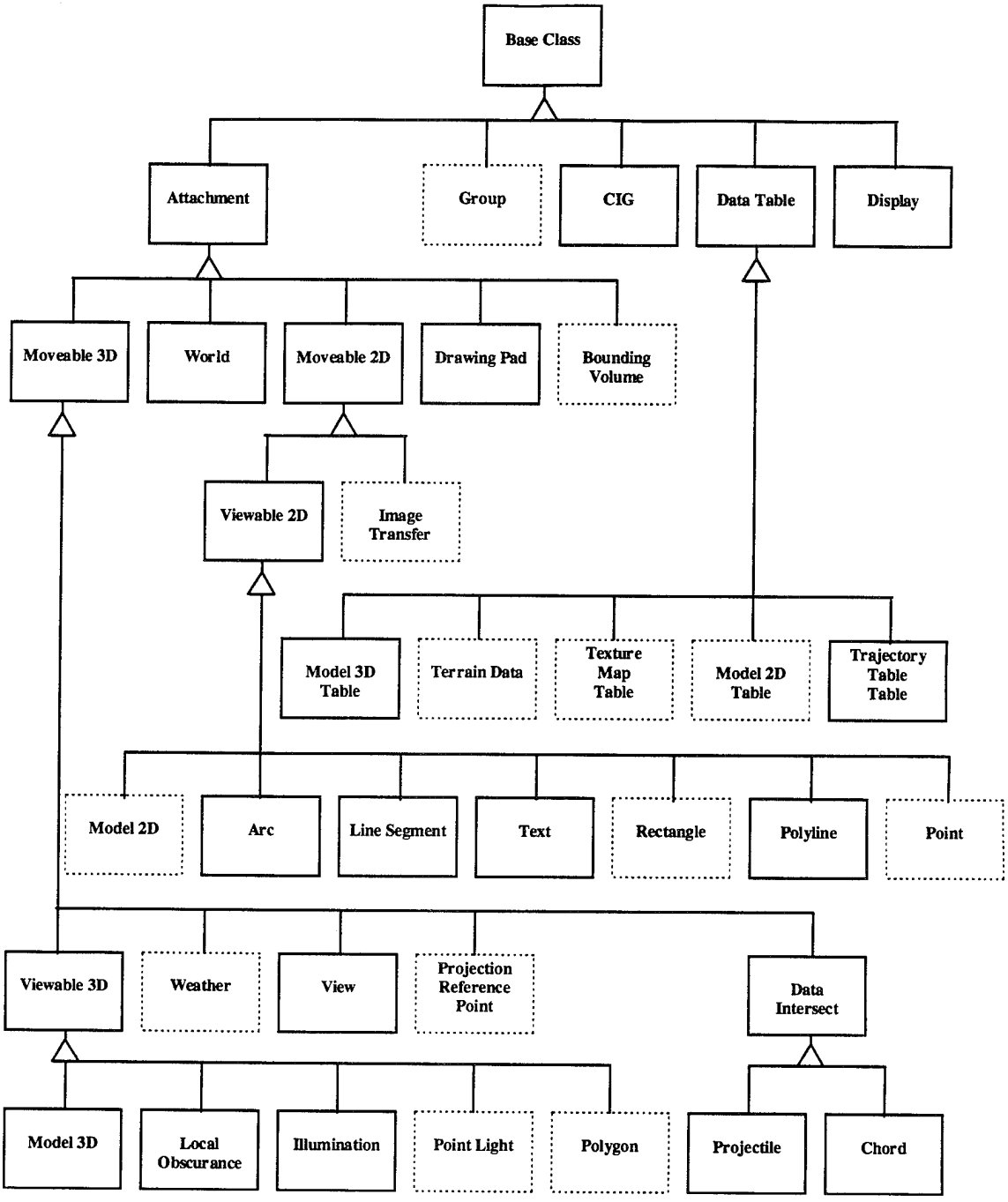

Figure 26. Clip Class Hierarchy (dashed lines indicate not implemented as of 9/20/95) [IDD94, 3-4]

3.3.1.2 The Clip Architecture

It would be inaccurate to label Clip an "architecture" using the same definition as applied to ObjectSim and Easy_Sim. These latter two impose a structure on conforming applications, whereas Clip merely provides an abstract interface. As mentioned in the previous subsection, however, all Clip simulations include instances of key Clip classes. To an extent, then, the usage of Clip does imply a certain structure for the applications.

3.3.1.3 Platform-Independence

The Clip approach allows the development of simulation applications which do not rely on the CIG. This is because all aspects of the image generation are accessible by manipulation of the standard Clip classes. Regardless of how these classes are implemented from one platform to another, applications always work through the same interface.

In an email message from the Advanced Distributed Simulation Division [Jan95a], Janett asserted that Loral actually demonstrated this platform-independence:

> Using Clip we have proven that a single application can drive two completely different image generators (SGI vs GT200) with no code changes to that application.

In that same message, Janett also said that porting the application side of Clip wasn't difficult. Apparently there is little platform-dependence even in the implementation of the code which comprises the Sim process. An implication of this inference is that it might not be overly difficult to reimplement the application side of Clip in Ada 95.

3.3.1.4 Clip and Ada

As of September of 1995, there was no way to write Ada applications using the Clip interface. Since using Ada was an important objective of this thesis, possible solutions to this problem were examined. Three options were considered:

- Write an Ada binding (or bindings) to the Clip libraries on the application side of Clip. Further research would have been required to determine the most appropriate positioning of the binding(s). Figure 27 shows several alternatives:

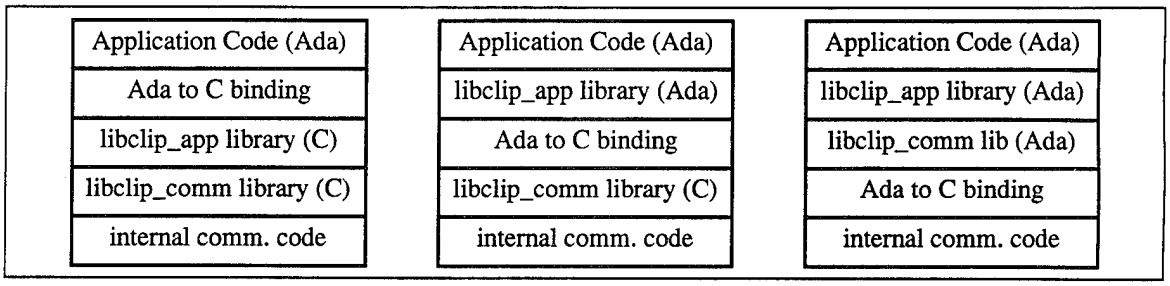

Figure 27. Potential Ada bindings to Clip

(Loral, also interested in offering the capability to develop Ada Clip applications, pursued the approach shown in Figure 27's leftmost box, using Ada 83.)

- Replace the entire Sim process code with Ada. This would require rewriting some of the Vistaworks internal communication code, which is proprietary. This possibility is illustrated in Figure 28.

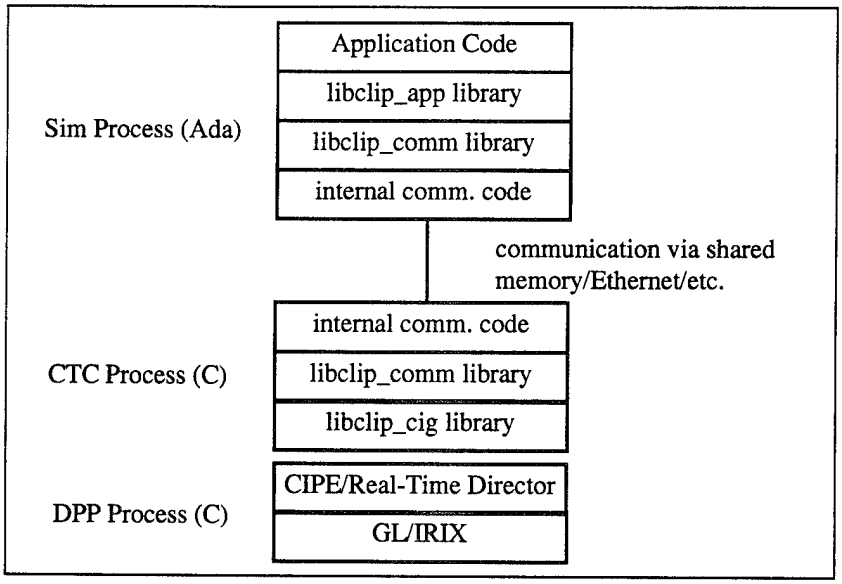

Figure 28. Replace the Sim process with Ada code

- Abandon all of the Vistaworks code. Reimplement the Clip interface in Ada 95. The result might be something like this:

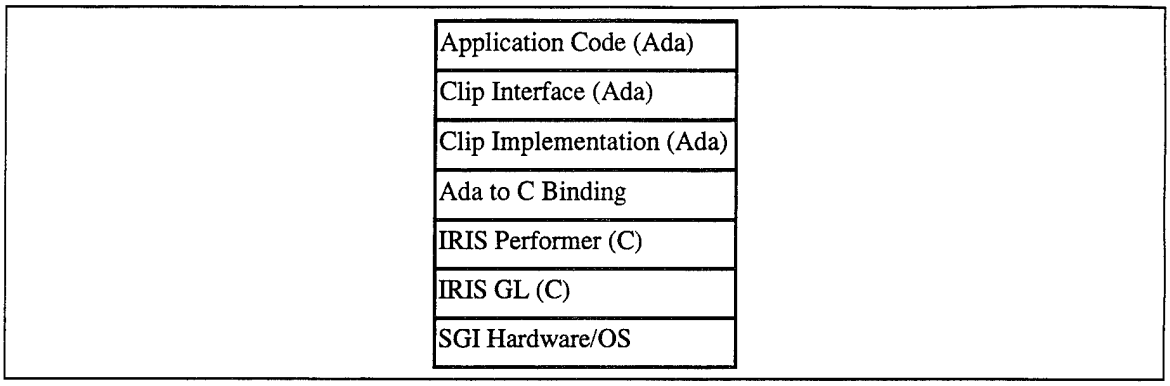

Figure 29. A possible new Clip implementation

Because of its emphasis on platform-independence, the Clip specification appeared to be an excellent starting point for a joint effort between the AFIT Lab and Loral. Unfortunately, Loral was unable to contribute to such an effort. Without assistance from Loral's Advanced Distributed Simulation Division, it was doubtful whether any of the concepts depicted in Figures 27-29 could be implemented. This uncertainty resulted in Clip being removed from consideration as other avenues were considered for ObjectSim.

3.3.2 Vega

A second commercial product was also examined for potential use in this effort. *Vega*, from Paradigm Simulation, Inc., is described as "a high performance visual simulation toolkit" [VPG94, 3]. In the *Developer's Configuration*, Vega provides an Application Progamming Interface (API) as a layer above Performer. Simulations predominantly make calls into Vega, but also have access to SGI's underlying software. Figure 30, an adaptation of Figure 1-1 in the Vega Programmer's Guide, shows this scenario.

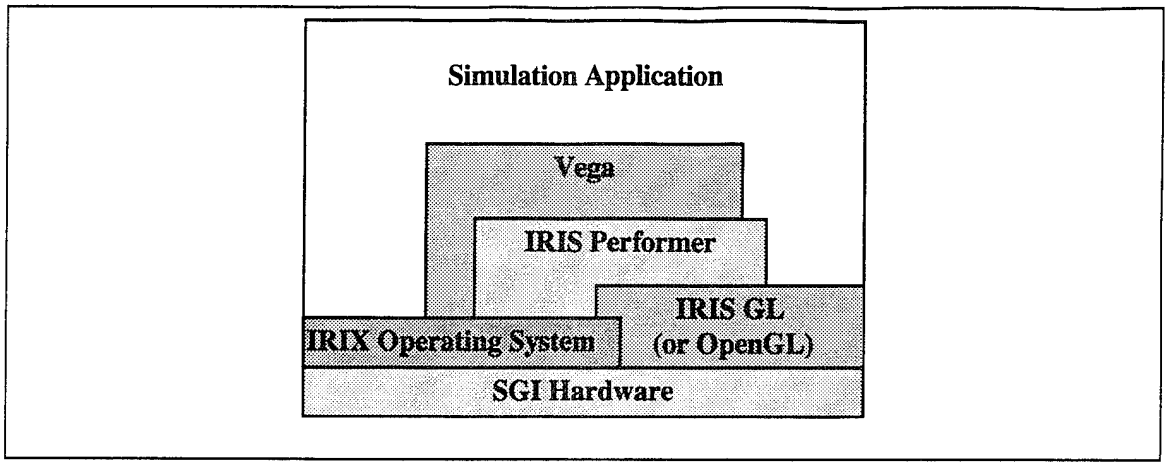

Figure 30. Vega's Developer's Configuration (adapted from [VPG94, 4])

An alternate configuration of Vega (known as the *Basic System*) allows the development of simulations which work strictly through the Vega API. In this scenario, simulations are not given access to the layers below Vega (i.e., Performer, GL, and the IRIX operating system). Simulations run separately from a Vega run-time process. Communication occurs via shared memory. This setup, shown in Figure 31, is quite comparable to the Clip approach (see Figure 24).

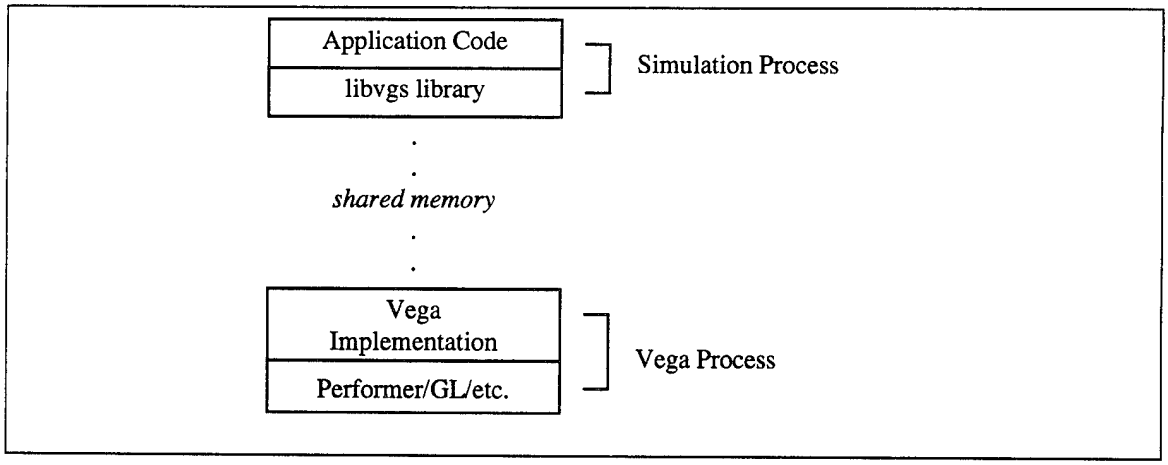

Figure 31. Vega's Basic System Configuration

3.3.2.1 Vega Classes

In either configuration, Vega allows simulation applications to take advantage of Vega (and Performer) services via manipulation of an object-oriented interface. The key classes in this interface are:

- **Window** - where rendering occurs; can contain any number of channels; a video display can have any number of windows
- **Channel** - describes both a view into a scene as well as a rectangular viewing region within a window
- **Observer** - a viewpoint within a scene; multiple channels may be driven by a single observer (example: an infrared channel and a "daylight" channel may be two ways of looking at a scene from the same viewpoint)
- **Scene** - the collection of everything viewable within the simulation
- **Object** - a viewable entity
- **Player** - essentially a moving local coordinate system within a scene; within the local coordinate system may be any number of visible objects (example: an F-16 and a number of missiles); may be positioned relative to another player, an observer, or the scene itself

Figure 32 shows the aforementioned classes and their interrelationships.

3.3.2.2 Vega and Ada

Paradigm does not offer an Ada implementation of the Vega API. An email conversation with a representative from the company revealed Paradigm is not currently planning to provide an Ada interface in the future [Cur95].

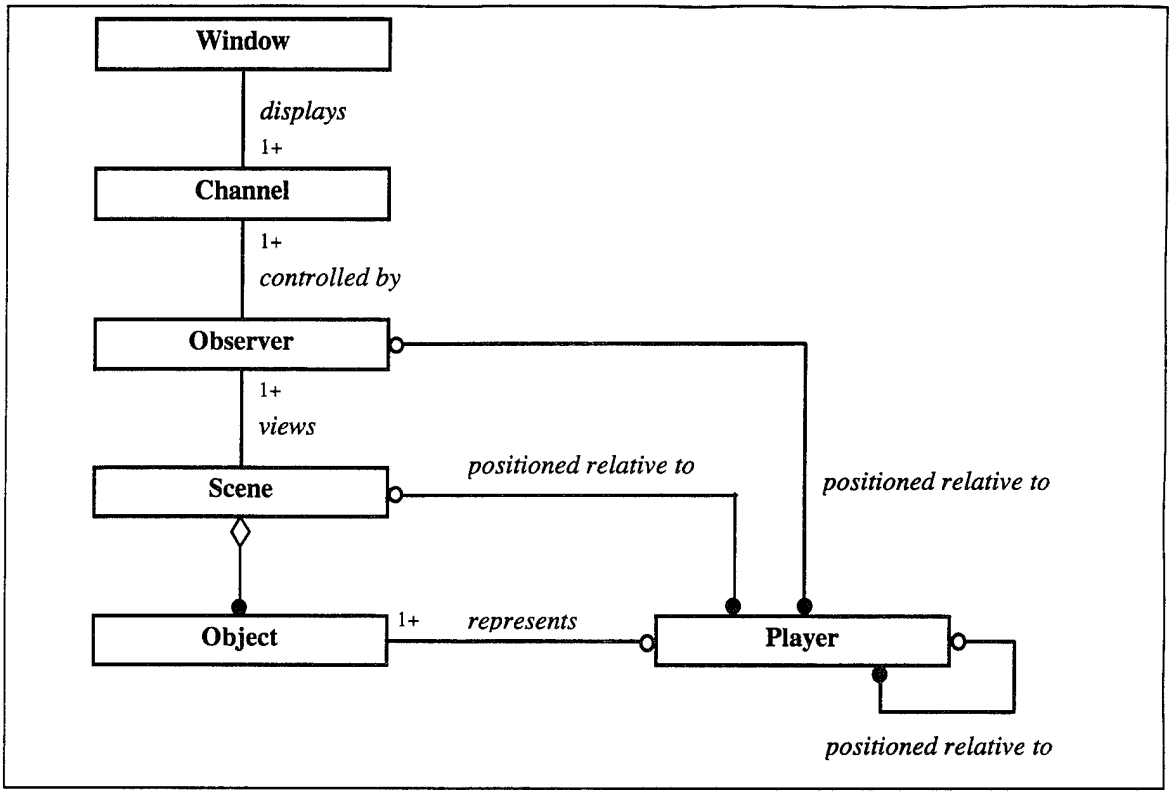

Figure 32. Class relationships in Vega

3.3.2.3 Platform-Independence

There is a direct correlation between Vega's class organization and SGI's software hierarchy. Specifically, the **Channel** and **Scene** classes correspond to Performer constructs. The **Window** class describes a GL window. At the core, Vega's abstraction of a visual simulation is the same as SGI's, as implemented by GL and Performer.

Vega does add higher-level concepts such as observers, objects and players, but these concepts are built on a foundation which is very tightly coupled to the Performer library. In fact, in the opening chapter of the Vega Programmer's Guide, Paradigm touts the

"Vega-Performer Connection" and the close relationship between Paradigm and SGI [VPG94, 4].

Vega offers an interface at the level of abstraction desired by the AFIT Lab. The degree of coupling to a specific rendering library (Performer), however, precluded the adoption of this API as part of an overall platform-independent visual simulation software architecture.

3.4 Adapt Easy_Sim

The route selected for the development of ObjectSim 3.0 was to adapt Easy_Sim to address the two issues cited in section 1.2: lack of a distributed simulation capability and platform dependencies. The next two subsections discuss these two issues, and the final subsection of this chapter describes the high-level design of ObjectSim 3.0.

3.4.1 Distributed Simulation

The first reference point for how to incorporate a distributed simulation capability into the existing Easy_Sim architecture was the ObjectSim framework. Although the ObjectSim architecture did not originally account for participation in distributed simulations, the C++ framework ultimately included classes which assisted applications in this regard. As reported in section 2.6.1, this portion of the framework can be confusing. However, analysis revealed one important fact: an ObjectSim application which participates in a distributed simulation communicates with an independent network management process, called the Object Manager, as shown in Figure 33.

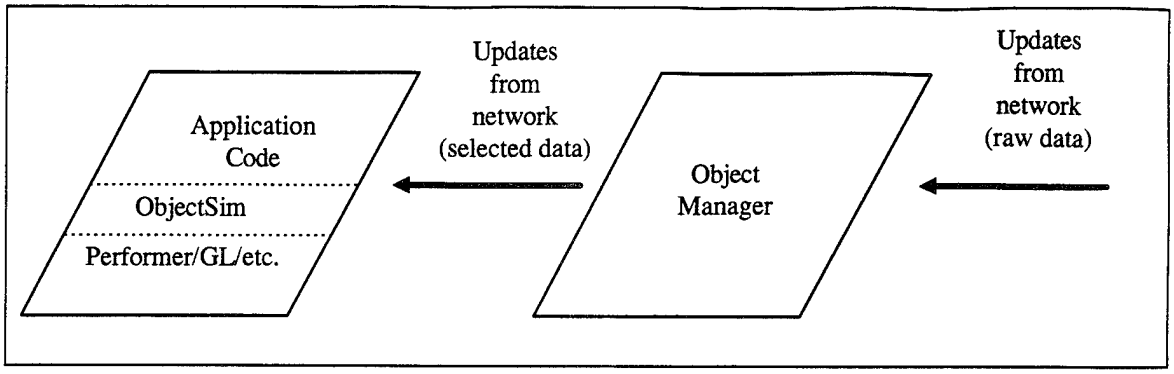

Figure 33. An ObjectSim application works with an Object Manager process to participate in a distributed simulation.

Figure 33 suggests that adjusting the Easy_Sim architecture to allow for distributed simulations would involve expanding the architecture to encompass multiple processes. This prospect is reinforced by the fact that both Loral and Paradigm use multiple-process models for some configurations of their products. The analyses of ObjectSim, Clip and Vega inspired consideration of a new 3-process architecture, as shown in Figure 34.

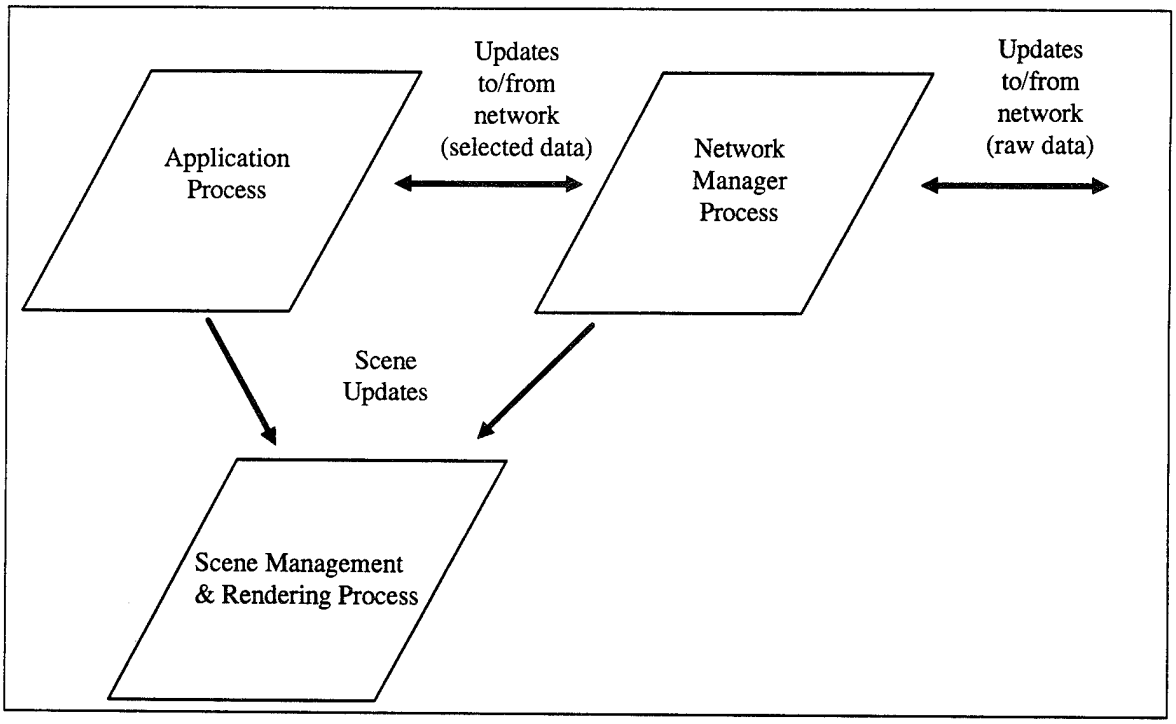

Figure 34. A possible 3-process model for distributed simulation

The design shown in Figure 34 is similar to that shown in Figure 33, with the added concept of a separate process which manages and renders the scene, based on updates from both the application and the network manager. This "scene management" process would be the only process to make calls to the platform-specific graphics and simulation libraries, such as GL and Performer.

This realization raised the issue of whether this thesis effort was pursuing objectives at two different levels of architectural abstraction. One objective, how to allow for distributed simulations, involved higher-level issues such as how many processes Lab simulations should be composed of, what the interfaces to these processes should look like, and how the processes would communicate with each other. The second objective, independence from platform-specific libraries, seemed to be at a lower level of abstraction, and of interest to only one of the three proposed processes.

The decision was made to focus on the issue of platform-independence and to drop further consideration of the distributed simulation problem.

3.4.2 Platform-Independence

As discussed in section 2.7.9, the Easy_Sim architecture is, for the most part, conceptually independent of any platform. The problem is that the application frameworks reference Performer constructs at both the specification and implementation level, and do not discourage applications from using these same constructs. Although no substantial simulation applications using Easy_Sim have been written, examination of existing

ObjectSim applications shows that applications rely heavily on these constructs. Table 2 shows the number of Performer references found in four ObjectSim applications, as well as the approximate number of lines of source code in each. More detailed information is offered in Appendix A.

Application	Performer References	Lines of Code
Battle Bridge	2394	19915
Space Modeler	1992	16948
Virtual Cockpit	3420	24520
Debriefing Tool	1686	24515

Table 2. ObjectSim application Performer dependencies

Because it is exactly the Performer references noted in Table 2 which hinder application portability, eliminating these references became a crucial goal during the design of ObjectSim 3.0. In fact, ensuring applications rely solely on ObjectSim is a cornerstone of the updated architecture, which is presented in the next subsection.

3.4.3 ObjectSim 3.0

The primary difficulty faced during the design of ObjectSim 3.0 was how to improve the portability of simulation applications without denying those applications the services provided by GL and Performer. The chosen solution involved widening the definition of the architecture to include lower-level services previously provided by these platform-dependent libraries. This fundamental change is depicted in Figure 35.

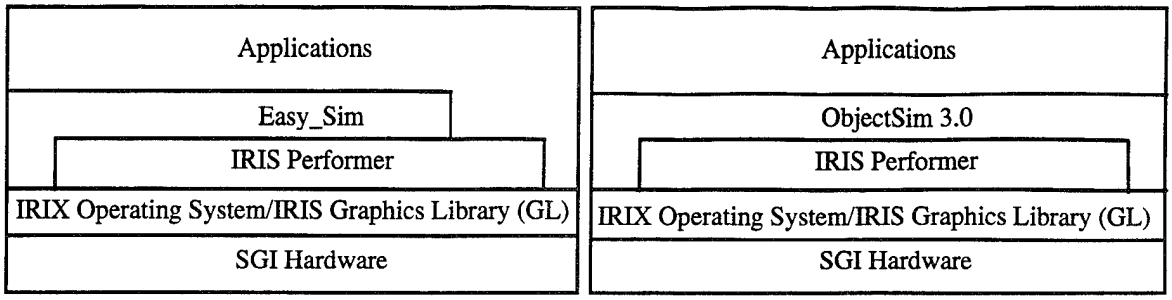

Figure 35. Software layering with Easy_Sim vs. ObjectSim 3.0

The ObjectSim 3.0 layer shown in Figure 35 is divided into two parts: ObjectSim Low-Level Services (LLS) and the ObjectSim Application Programming Interface (API). These two parts are discussed in the next two subsections.

3.4.3.1 Low-Level Services

The intention of ObjectSim Low-Level Services is to offer application developers a collection of capabilities which are typically needed for visual simulations, but which have historically been provided by platform-specific libraries. For example, as evidenced by Table 2, applications previously developed at AFIT have used the Performer library extensively. Furthermore, discussions with Lab developers and some additional research revealed a need for access to the resources of the IRIS Graphics Library as well. The key issue during the design of the LLS thus became how to ensure continued access to these libraries without allowing direct references from the applications. The result was a close correlation between the LLS and SGI's Performer and GL libraries. This close correlation means an understanding of the SGI software essentially implies an understanding of the ObjectSim Low-Level Services.

The ObjectSim LLS presents a set of libraries which offer the same functionality as the SGI libraries. ObjectSim applications can therefore access the extensive features of the SGI software without calling those libraries directly. While at first glance this may seem to reduce platform-independence in name only, consider the benefits of this arrangement:

- By obviating application references to Performer and GL, the LLS isolates platform-dependencies on these libraries **within itself**, giving the Lab a measure of control over these reliances.

- Since the Lab controls the interface to Low-Level Services, the names of constants, types and subprograms (and their parameters) are chosen by the Lab. The LLS maps these names to their SGI counterparts.

- The LLS can make upgrades to the GL and Performer libraries transparent to using applications. Features which might be dropped in new releases could conceivably be provided by the LLS itself. Adjustments to other GL and Performer capabilities could be compensated for by the LLS, reducing or eliminating any impact on the applications. And the Lab controls the integration of new GL and Performer features into the applications.

So although the ObjectSim 3.0 Low-Level Services components do not put the Lab in a position to declare true independence from the platform-specific SGI libraries, they collectively form a software layer which shields applications from direct dependencies on those libraries. This "step in the right direction" was subsequently implemented in Ada 95. The discussion of that implementation follows in chapter 4.

Following are brief descriptions of the components of the LLS, including the SGI libraries to which each component is related.

- **Math_Utilities**: replicates the capabilities of the Performer "libprmath" library; includes functions such as Sin, Cos, Min, Max, vector and matrix operations

- **Rendering**: replicates the capabilities of the Performer "libpr" library; includes functions such as Set_Texture_Format, Set_Shininess, and Set_Color

- **Rendering-Stats**: consists of the five statistics-related constants in the Performer "libprstats" library

- **Vis_Sim**: replicates the capabilities of the Performer "libpf" library; includes functions such as Initialize_Pipe, Attach_Channel_To_Scene and Set_Viewport

- **Vis_Sim-Stats**: replicates the capabilities of the Performer "libpfstats" library; includes functions to selectively draw channel statistics on-screen

- **Vis_Sim-Utilities**: replicates the capabilities of the Performer "libpfutil" library; includes functions related to keyboard and mouse input, as well as certain texture utilities

- **Vis_Sim-Import_Utilities**: offers data import capabilities from the Performer "libpfsgi" and "libpfflt" libraries

- **Windows**: offers a subset of the windowing capabilities of the GL library; a limited set of functions are presently specified

3.4.3.2 Application Programming Interface

Although the original ObjectSim architecture and the successor Easy_Sim architecture differ in what services they offer to application developers, both share a common goal: to provide at least some capabilities at an even higher level of abstraction than those provided by Performer. By offering such features, these architectures reduce the need for applications to use Performer's lower-level services. The same can be said of the Clip and Vega APIs discussed earlier in the chapter.

This observation indicated that additional platform-independence gains could be made by constructing a second layer on top of the LLS. If applications could rely primarily on this

second layer, they would be less likely to call upon Low-Level Services itself. The result would be an overall reduction in the number of dependencies from the applications to the LLS. Transitively, this would mean lower coupling between each application and the GL and Performer libraries.

Thus, a higher-level services layer was designed. In keeping with the terminology of the commercial products, this layer was designated the *ObjectSim 3.0 Application Programming Interface (API)*. The goal of the API is to simplify the development of visual simulations by abstracting away selected parts of the ObjectSim LLS and providing a set of higher-level routines which developers would otherwise end up writing from scratch.

Like Clip and Vega, the ObjectSim API is object-oriented. The key classes are:

- **Scene**: an entire viewable "world", with its own coordinate system, an optional environment, and any number of entities
- **Entity**: anything which might be placed within a scene, such as a light or a 3d model
- **Environment**: the "surroundings" within a scene, such as ambient light, background color and fog
- **View**: a movable viewing frustum; every view has an associated scene, but the view is considered external to the scene, not a part of it
- **Renderer**: responsible for actually drawing the views on a display; since only one of these is ever desired, **Renderer** is not really a class--it's a single object; (The box representing the **Renderer** in Rumbaugh object diagrams is shaded gray to distinguish it from regular classes. This is a convention adopted for illustrative purposes in this thesis.)

The relationships amongst these classes are captured in the Rumbaugh object diagram of Figure 36. Note the subclasses of **Entity**. This inheritance hierarchy is explained later in the section.

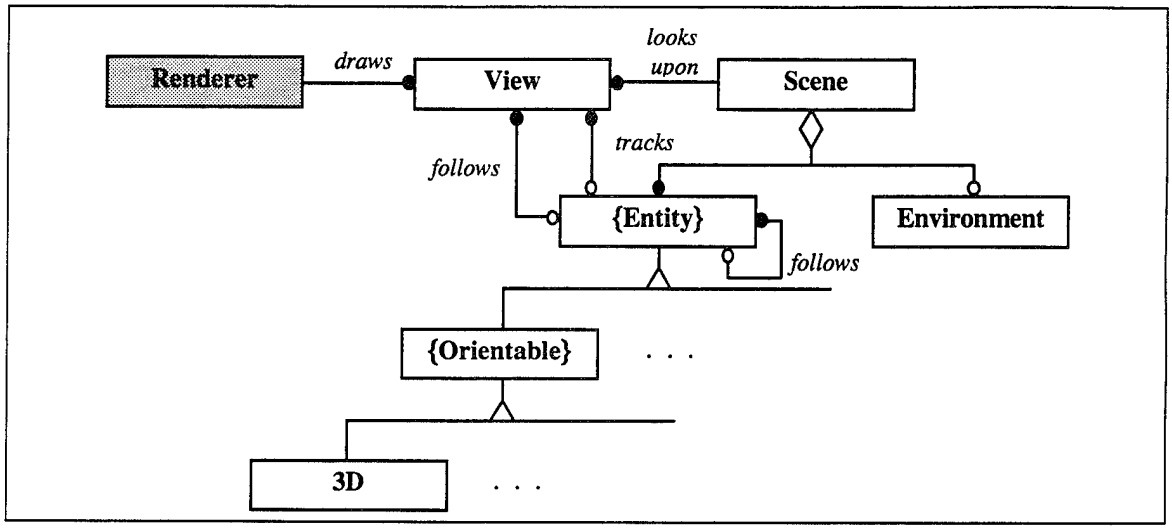

Figure 36. ObjectSim's object-oriented API

The ObjectSim API was developed with an important underlying principle--simplicity. In an effort to make the API easy to learn and understand, the interface was designed with a strict one-way hierarchy of class dependencies. No two classes are mutually dependent at the specification level. For example, the **Renderer** "understands" and manipulates instances of the **View** class, but **View** instances never call on the services of the **Renderer**. Similarly, the **Scene** class manipulates entities via calls to the **Entity** class, but **Entity** instances have no notion of a scene (or a view, or the **Renderer**). The complete hierarchy is shown in Figure 37. Each arrow represents a sort of client-server relationship, with the arrow pointing towards the server.

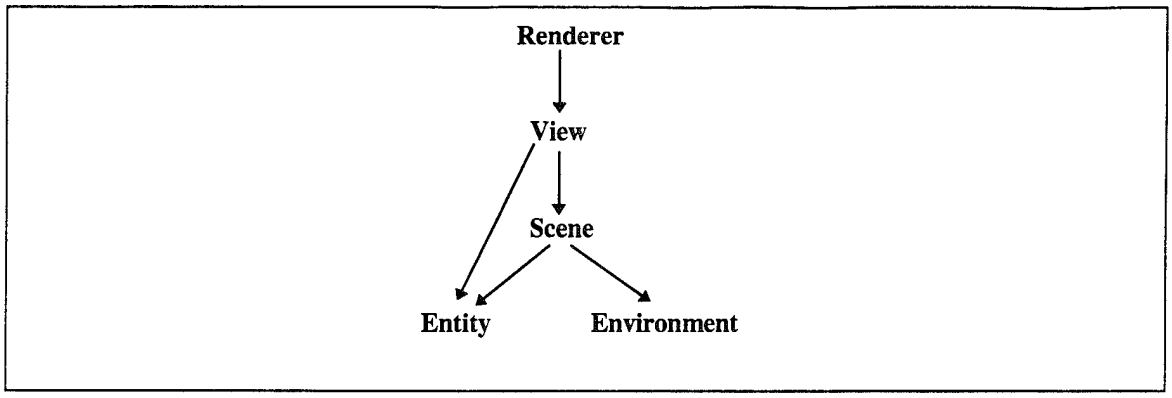

Figure 37. The ObjectSim 3.0 API class dependency hierarchy

The following subsections describe each of the API classes in greater detail. Most descriptions include a Rumbaugh diagram to show the public attributes and methods of the classes. Note the following conventions for these diagrams:

- Constructor and destructor operations are available in all cases, but are not shown in the diagrams.
- Operations for retrieving and adjusting attribute values are available in all cases, but are not shown in the diagrams.
- Explanatory Ada-style comments are given to the right of each diagram.

3.4.3.2.1 Scene

The **Scene** class is the most complex of the ObjectSim 3.0 API, which is not surprising, since it represents the abstraction of an entire viewable world. In the simplest case, a scene is just an empty coordinate system with a black background. The background is altered via the **Environment** class, described in subsection 3.4.3.2.3. A scene has user-definable boundaries which establish a three-dimensional, brick-shaped, scene volume.

In general, there will be some number of entities positioned within a scene's coordinate system. The positioning of entities is accomplished by calls to the **Scene** class. In addition to positioning commands, developers can direct entities to "follow" one another. As the lead entity is moved, the follower is moved automatically. Multiple entities can follow a single leader, maintaining different relative positions. Also, one entity can follow another entity, which is following a third entity, and so on.

Figure 38 depicts the **Scene** class.

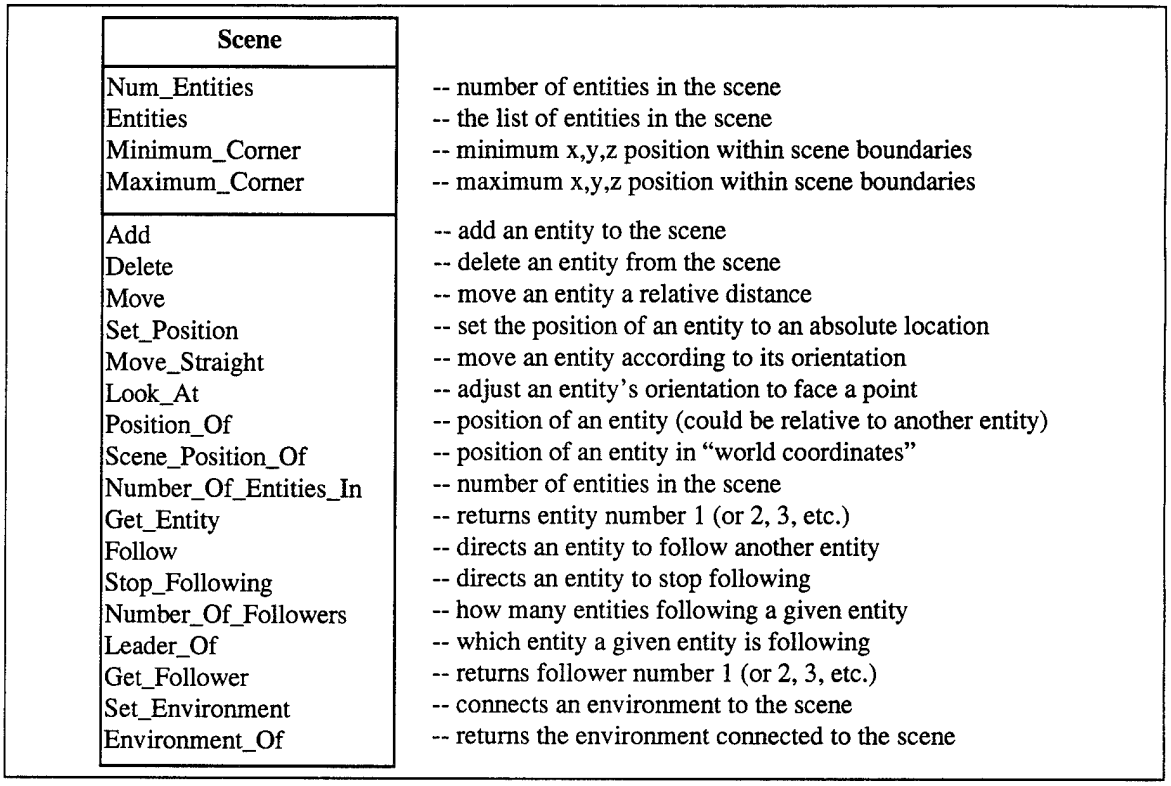

Figure 38. The ObjectSim 3.0 API Scene Class

3.4.3.2.2 Entity

An instance of the **Entity** class is anything which can be positioned in a scene. Although the ObjectSim API establishes that every entity has a scene position, this information is

stored as part of the scene, not as part of the entity. This approach differs from the original ObjectSim and Easy_Sim approaches. The advantage is that, in ObjectSim 3.0, entities are completely oblivious of the fact that they are part of a larger scene. In fact, the **Entity** class has no visibility to, or understanding of, any other API classes. This was done intentionally as part of the effort to keep the API simple, as described in section 3.4.3.2.

It should also be noted that the **Entity** class differs from the **Player** classes of the earlier architectures. **Everything** which might be positioned within a scene, including the terrain and stationary objects, is considered an entity in ObjectSim 3.0. In the original ObjectSim and Easy_Sim architectures, the terrain and stationary objects came under the heading of **Terrain** or **Environment**. This conceptual difference inspired the name change from "player" to the more generic term "entity".

The **Entity** class is abstract, meaning there will be no simulation participants with the generic "entity" designation. Rather, participants must be members of some particular refinement of the **Entity** class. Only one such refinement has been specified at this time, and that subclass is described in the next subsection.

The **Entity** class has no public attributes or methods, so the Rumbaugh diagram is omitted.

3.4.3.2.3 Orientable

The **Orientable** class is the only currently-defined subclass of **Entity**. Members of this subclass are distinguished from other entities by the fact that they have an associated heading, pitch and roll. Examples include three-dimensional models and directable light sources. (A non-directable light source is one example of a non-orientable entity.)

Orientable is an abstract subclass. The attributes are listed in Figure 39, but the operations which retrieve and adjust those attributes are omitted, according to the convention of this section.

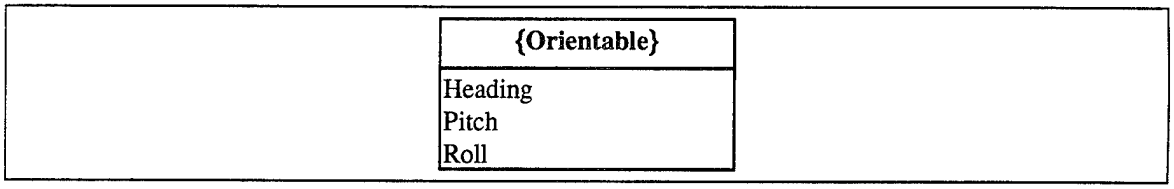

Figure 39. The ObjectSim 3.0 API Orientable Entity Class

3.4.3.2.4 3D

The **3D** subclass is the only currently-defined concrete descendant of **Entity**. Presently, then, any entity in an ObjectSim 3.0 simulation will be a member of this subclass. The only operation of this class is to read the geometry of the three-dimensional model from a file. (Supported file formats are the same as those supported by Performer 1.2.) The **3D** subclass is shown in Figure 40.

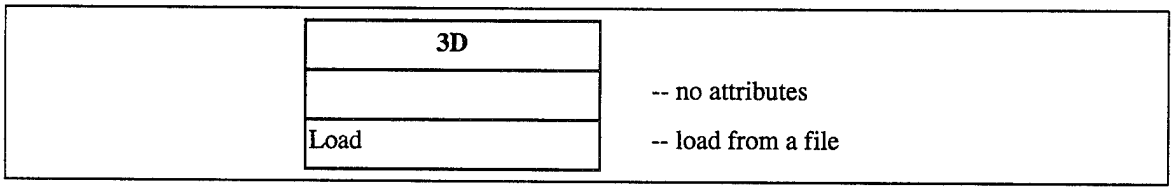

Figure 40. The ObjectSim 3.0 API 3D Model Class

3.4.3.2.5 Environment

The **Environment** class deals with aspects of a scene which cannot come under the heading of "entity." For example, if a view is positioned at the outer edge of a scene, looking **away** from the scene, what does it see? Perhaps nothing--just blackness. Or pehaps a blue sky and a dull, brown terrain. The ObjectSim API mandates that if a scene has no associated environment, the overall scene background will default to black. If a scene **does** have an associated environment, attributes of the **Environment** class allow the background color to be changed. These are the only class attributes currently specified. It is anticipated that the **Environment** class will be expanded in the future to account for such things as fog and time-of-day.

The **Environment** class is depicted in Figure 41. Again, attribute read/write operations have been omitted.

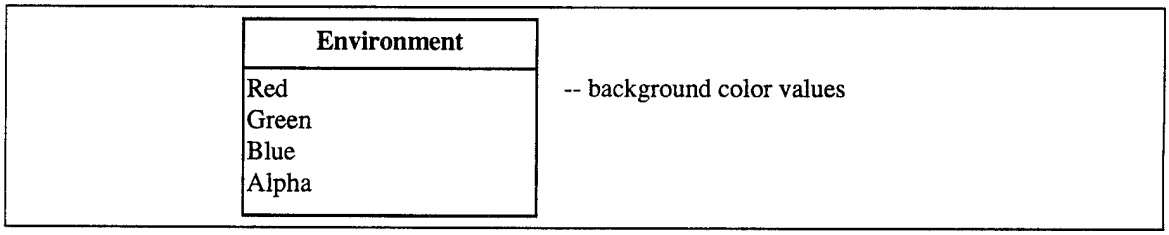

Figure 41. The ObjectSim 3.0 API Environment Class

3.4.3.2.6 View

The **View** class provides an abstraction similar to the Performer concept of a channel. (See [Har94] chapter 4.) A view includes a viewing frustum, which leads to four class attributes: the near and far clipping plane distances, and the horizontal and vertical fields-

of-view. These four attributes define the frustum. Furthermore, each view has a position within its associated scene, and an orientation which dictates which way the view is directed.

As for operations, a view can be directed to mimic another view, alter its orientation to "look at" a particular point in the scene, or follow an entity as it moves. A view can also be instructed to "track" a particular entity, which means the view will stay in place but continually look at the entity as the entity moves. Following and tracking can be combined.

The attributes and operations of the **View** class are summarized in Figure 42.

View	
Horizontal_FOV	-- horizontal field-of-view
Vertical_FOV	-- vertical field-of-view
Near_Clipping_Plane_Dst	-- distance to near clipping plane
Far_Clipping_Plane_Dst	-- distance to far clipping plane
Position	-- position of view within scene
Orientation	-- heading, pitch and roll
Move	-- move a relative distance
Scene_Position_Of	-- view position in "world coordinates"
Follow	-- follow an entity
Stop_Following	-- stop following an entity
Leader_Of	-- the entity being followed
Look_At	-- adjusts orientation of a view to face a point
Track	-- constantly change orientation to face an entity
Stop_Tracking	-- stop tracking an entity
Mimic	-- mimic another view
Set_Scene	-- connects a scene to the view
Scene_Of	-- returns the scene connected to the view

Figure 42. The ObjectSim 3.0 API View Class

3.4.3.2.7 The Renderer

The **Renderer** is a special "simulation executive" object which has a variety of simulation responsibilities. Only a small set of services have been identified at this point, but it is expected that this set will substantially expand in the future. Every simulation application will have exactly one **Renderer**, so it is not a typical class which can be instantiated. Instead, applications simply refer to "the Renderer." Developers call it to accomplish any pre-simulation initialization, set the frame rate, and draw the frames. Furthermore, a current simplifying assumption is that the **Renderer** draws to a single implied screen. This screen can be subdivided into multiple viewports, and these viewports are manipulated via calls to the **Renderer**. The **Renderer** "understands" what a view is, and is capable of directing each instance of the **View** class to one or more viewports. Attributes and operations are shown in Figure 43.

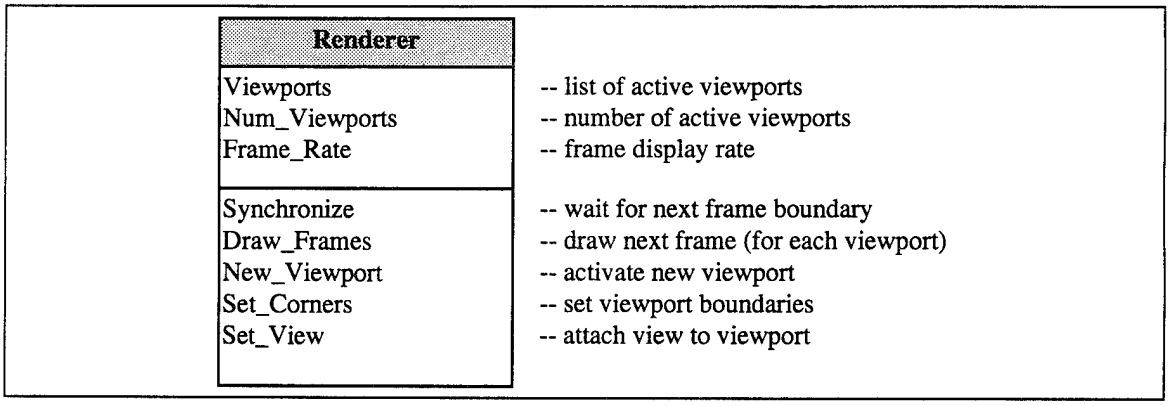

Figure 43. The ObjectSim 3.0 API Renderer Object

3.4.3.3 Application Framework

Given Low-Level Services and the Application Programming Interface, the next issue was how to transform ObjectSim 3.0 into an "architecture" like its predecessors. Considera-

tion of this issue raised several questions, such as: What is the relationship between an API and an architecture? Why are commercial offerings typically of the first category, while the AFIT Lab continues work in the latter realm? What are the advantages and disadvantages of the two approaches?

A natural first step towards answering these types of questions was to compare the Easy_Sim architecture (see Figure 13) to the new ObjectSim 3.0 API (Figure 36). There are clearly similarities. The architecture and the API both include the notions of an "environment" and a "view," and the Easy_Sim **Player** class seems roughly comparable to the ObjectSim **Entity** class.

A key difference is Easy_Sim's overt mandate of a particular structure for any Easy_Sim simulation. Each simulation must include an instance of some subclass of the architecture's **Simulation** class, as well as **Player_Manager**, **View_Manager**, and **Model_Manager** instances. In this way, the architecture establishes the basic design of every Easy_Sim simulation.

In contrast, as shown in Figure 36, the ObjectSim API does not stipulate any structure for simulations which use it. An ObjectSim API simulation could be a single procedure which manipulates instances of the various API classes. (All simulations which take advantage of the API **will** have certain things in common, however. Namely, every simulation will

have exactly one **Renderer** object, some number {usually just one} of scenes, possibly an environment, some number of views, and some number of entities.)

To take ObjectSim 3.0 a step closer to Easy_Sim, there is no reason why every ObjectSim 3.0 simulation could not be required to have "manager" objects to administer the scenes and views. This line of reasoning yields Figure 44.

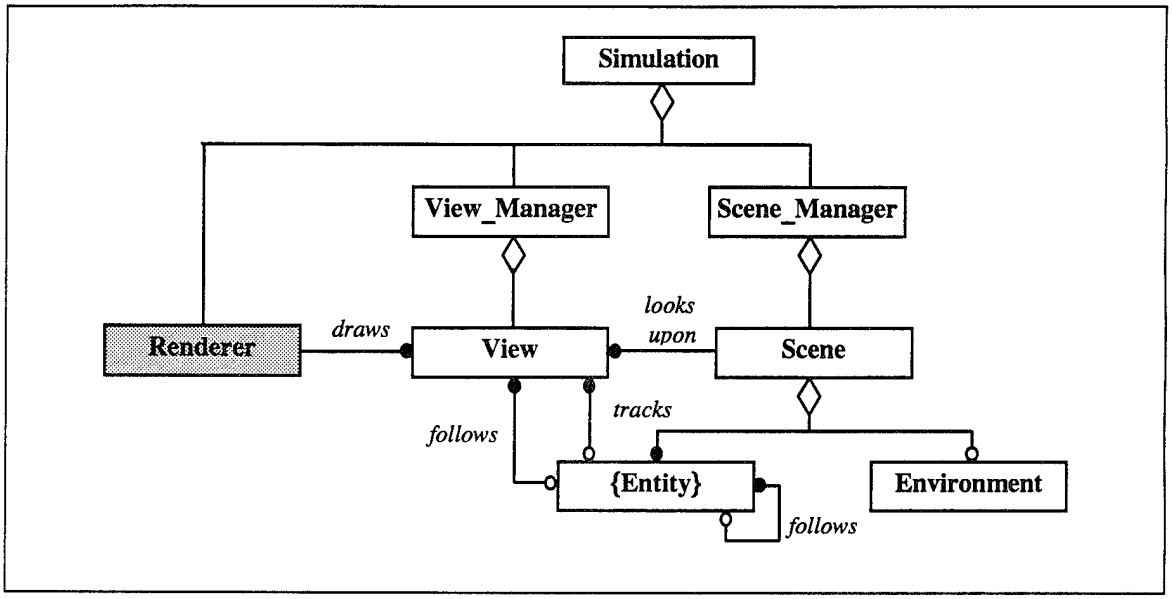

Figure 44. The API becomes an architecture?

The Rumbaugh diagram of Figure 44 certainly appears to qualify as an architecture, though there remains one important difference when compared to Easy_Sim. The Easy_Sim **View**, **Player**, **Modifier** and **Environment** classes are abstract, and they are abstract in essentially the same way. Each class includes an **Update** operation which specifies the behavior of class instances at run-time. Developers tailor Easy_Sim by designing new classes with particular behaviors. For example, in an Easy_Sim simulation with two F-15 models flying in different patterns, there would be one instance of each of

two **Player** subclasses, where the only difference between those subclasses is the **Update** operation which details the motion of class instances.

Using the ObjectSim 3.0 API, this same simulation would be accomplished by declaring two variables, both of the **3D** class. The application would move the F-15s via calls to the **Scene** class, and these calls could appear anywhere in the application. Thus, unlike in the preceding approach, the behavior of the F15s is not necessarily localized to one particular place in the source code.

Research into this fundamental difference in operation revealed apparent recognition of **both** concepts by the object-oriented community. White papers ([Tal93] and [Tal94]) from Taligent, Inc., a company dedicated to object-oriented technology, validates the usefulness of both APIs and architectures. Interestingly, Taligent classifies both concepts under the heading of "frameworks." From page 4 of [Tal94]:

> Another categorization [of frameworks] is based on how a framework is used--whether you derive new classes or instantiate and combine existing classes. This distinction is sometimes referred to as architecture-driven versus data-driven, or inheritance-focused versus composition-focused.
>
> Architecture-driven frameworks rely on inheritance for customization. Clients customize the behavior of the framework by deriving new classes from the framework and overriding member functions.
>
> Data-driven frameworks rely primarily on object composition for customization. Clients customize the behavior of the framework by using different combinations of objects. The objects that clients pass into the framework affect what the framework does, but the framework defines how the objects can be combined.

Which is the better approach? As is often the case, arguments can be made in favor of either technique. Taligent offers the following insight:

> Frameworks that are heavily architecture-driven can be difficult to use because they require clients to write a substantial amount of code to produce interesting behavior. Purely data-driven frameworks are generally easy to use, but they can be limiting [Tal94, 4].

Apparently, vendors such as Loral and Paradigm have chosen the API (or data-driven framework) concept for its ease of use, whereas the AFIT Lab has historically worked in the realm of inheritance-focused architectures (or architecture-driven frameworks).

For the sake of comparison, the ObjectSim 3.0 API was altered to form the ObjectSim 3.0 Framework. The API and the Framework are virtually identical in capability, but the API is tailored via composition, whereas the Framework is tailored via inheritance. The design of the Framework is exactly that shown in Figure 44, with the caveat that the **Scene**, **Entity**, **Environment** and **View** classes have new, abstract **Update** operations. Developers specify the behavior of each **Scene, Entity, Environment** and **View** instance by writing a new **Update** operation for every subclass of these classes. The ObjectSim Framework calls **Update** for every object instance during each frame of the simulation. The result of having both an API and Framework is that an ObjectSim 3.0 simulation may take advantage of the API, with minimal structural requirements on the simulation itself, or it may choose to use the Framework, in which case the simulation becomes a specialized instance of Figure 44. The differences between these two approaches is easiest to see by way of example, which is a key part of chapter 4.

4. Specification and Implementation in Ada 95

4.1 Overview

Section 1.3 of this thesis mentioned a desire to continue demonstrating the viability of the Ada programming language in the visual simulation domain. To that end, the ObjectSim 3.0 LLS, API and Framework outlined in chapter 3 were implemented in Ada 95. Although the current implementation does not meet 100% of the specifications, the majority of code is in place. Several demonstration programs have been completed. Chapter 4 is organized around the most straightforward of these demonstration programs.

4.2 A Simple Performer Program

Example 3-1 of [Har94] shows a very basic simulation application which uses SGI's Performer library. This sample application has been altered slightly and converted to Ada, and is presented as Figure 45. See [Har94, 36-41] for a complete explanation of this program.

Figure 45 shows the main program, **Simple**, and an embedded **OpenPipeline** routine. The main program begins by initializing and configuring the Performer library. It then loads some geometry from a file named "box.flt", and adds this geometry to the visual scene. After adding a light source to the scene as well, the program opens a Performer "pipe," which also causes an on-screen graphics window to be opened. The final step in setting up the simulation is the creation and configuration of a Performer "channel."

```
--
-- simple.adb - a simple Performer program
-- adapted from page 36 of the IRIS Performer Programming Guide
--
with GL;
with Performer_Pr;
with Performer_Pf;
with Performer_Pfflt;
with Performer_Prmath;
with Basic_Types; use Basic_Types;
with Interfaces.C.Strings; use Interfaces.C.Strings;
procedure Simple is

   Pipe    : Performer_Pf.Pfpipe;
   Scene   : Performer_Pf.Pfscene;
   Channel : Performer_Pf.Pfchannel;
   Root    : Performer_Pf.Pfgroup;

   Elapsed_Time : Float64 := 0.0;

   --
   -- OpenPipeline is called to open the viewing window and set up
   -- default view parameters.
   --
   procedure OpenPipeline (Pipe : Performer_Pf.Pfpipe) is

      LightModel    : Performer_Pr.Pflightmodel;
      Window_Number : GL.Window_Id;

   begin

      -- Negotiate with window manager
      GL.Foreground;
      Window_Number := GL.Winopen("Simple");
      GL.Winconstraints;

      -- Negotiate with GL
      Performer_Pf.PfInitGfx(Pipe);

      -- Maximize lighting
      LightModel := Performer_Pr.PfNewLModel(Performer_Pr.PfGetSharedArena);
      Performer_Pr.PfLModelAmbient(LightModel,1.0,1.0,1.0);

      -- Create and apply default texture environment
      Performer_Pr.PfApplyTEnv(Performer_Pr.PfNewTEnv
                               (Performer_Pr.PfGetSharedArena));

      -- Apply the light model
      Performer_Pr.PfApplyLModel(LightModel);

      -- Enable selected options
      Performer_Pr.PfEnable(Performer_Pr.PFEN_TEXTURE);
      Performer_Pr.PfEnable(Performer_Pr.PFEN_LIGHTING);
   end OpenPipeline;

begin -- Simple

   -- Initialize Performer
   Performer_Pf.Pfinit;

   -- Configure MP mode and start parallel processes
   Performer_Pf.Pfconfig;
```

```
   -- Read a single FLIGHT-format file
   Scene := Performer_Pf.Pfnewscene;
   Root  := Performer_Pfflt.Loadflt(New_String("box.flt"));
   Performer_Pf.Pfaddchild(Scene, Root);

   -- Configure and open pipeline
   Pipe := Performer_Pf.Pfgetpipe(0);
   Performer_Pf.Pfinitpipe(Pipe, OpenPipeline'Address);

   -- Get and configure channel
   Channel := Performer_Pf.Pfnewchan(Pipe);
   Performer_Pf.Pfchanscene(Channel, Scene);
   Performer_Pf.Pfchannearfar(Channel,1.0,1000.0);
   Performer_Pf.Pfchanfov(Channel,45.0,-1.0);

   -- Simulate for 20 seconds
   while Elapsed_Time < 20.0 loop

      declare
         S, C : Float32;
         View : Performer_Prmath.Pfcoord;

         Return_Value : Integer;
      begin
         -- Go to sleep until next frame time
         Return_Value := Performer_Pf.Pfsync;

         -- Compute new view position
         Elapsed_Time := Performer_Pr.Pfgettime;
         Performer_Prmath.Pfsincos(Float32(33.0 * Elapsed_Time),
                                   S'Address,
                                   C'Address);
         Performer_Prmath.Pfsetvec3(View.Hpr'Address,
                                    10.0 * S, -45.0 + 10.0 * C, 0.0);
         Performer_Prmath.Pfsetvec3(View.Xyz'Address,
                                    0.0, -50.0, 50.0);
         Performer_Pf.Pfchanview(Channel,View.Xyz'Address,View.Hpr'Address);

         -- Initiate cull/draw for this frame
         Performer_Pf.Pfframe;
      end;

   end loop;

   -- Clean up and exit
   Performer_Pf.Pfexit;

end Simple;
```

Figure 45. A simple Performer program

Once everything is set up, the main program goes into a 20 second loop. Within this loop, the viewpoint orientation is modified such that the target object (the box geometry read from the file) appears to move in a circle. Note the box itself never actually moves; the viewpoint is rotating about the box.

During the course of this approximately 110-line program, there are a few calls to the IRIS Graphics Library (e.g., identifiers preceded by "GL."), and many calls to IRIS Performer (e.g., identifiers preceded by "Performer_Pr", "Performer_Pf", etc.). The next section shows how the same program could be written strictly through use of ObjectSim Low-Level Services, without any direct calls to GL or Performer.

4.3 Low-Level Services

As explained in chapter 3, the goal of ObjectSim Low-Level Services is to shield applications from platform-specific graphics libraries. In this first implementation of ObjectSim 3.0, that goal has been accomplished via the process of *dependency-masking*. Dependency-masking means the LLS presents developers with the same conceptual abstractions as offered by GL and Performer, but with slightly different type, constant and subprogram names. For example, the specification of the Performer "pfChanScene" routine is:

```
procedure Pfchanscene (Chan  : System.Address;
                       Scene : System.Address);
```

The LLS version of this specification is:

```
procedure Attach_Channel_To_Scene (Channel : Channel_Type;
                                   Scene   : Scene_Type);
```

Granted, these are subtle changes. But, when used in place of direct references to Performer and GL, the LLS interface substantially changes the "feel" of a program. Figure 46 shows **Simple2**, an adaptation of the original **Simple** program which works directly with ObjectSim Low-Level Services. Notice there is no indication that GL or Performer is being used.

```
--
-- simple2.adb - a simple visual simulation
-- converted from simple.adb
--
-- This version of Simple works directly with ObjectSim.Low_Level_Services.
--
with Basic_Types; use Basic_Types;
with ObjectSim.Low_Level_Services.Math_Utilities;
with ObjectSim.Low_Level_Services.Rendering;
with ObjectSim.Low_Level_Services.Windows;
with ObjectSim.Low_Level_Services.Vis_Sim.Import_Utilities;
procedure Simple2 is

   use ObjectSim.Low_Level_Services;
   package Import_Utilities renames Vis_Sim.Import_Utilities;

   Elapsed_Time : Float64                := 0.0;
   Scene        : Vis_Sim.Scene_Type;
   Pipe         : Vis_Sim.Pipe_Type;
   Channel      : Vis_Sim.Channel_Type;
   Root         : Vis_Sim.Group_Type;

   procedure Open_Window (Pipe : Vis_Sim.Pipe_Type) is

      Window_Number : Windows.Window_Id_Type;
      Light_Model   : Rendering.Light_Model_Type;

   begin

      -- Negotiate with window-manager
      Windows.Foreground;
      Window_Number := Windows.Open("Simple2");
      Windows.Allow_Resizing;

      -- Negotiate with GL
      Vis_Sim.Initialize_Graphics(Pipe);

      -- Maximize lighting
      Light_Model := Rendering.New_Light_Model(Rendering.Get_Shared_Arena);
      Rendering.Set_Ambience(Light_Model,1.0,1.0,1.0);

      -- Create and apply a default texture environment
      Rendering.Apply_Texture_Environment
         (Rendering.New_Texture_Environment(Rendering.Get_Shared_Arena));

      -- Apply the light model
      Rendering.Apply_Light_Model(Light_Model);

      -- Enable selected options
      Rendering.Enable(Rendering.Enable_Texture);
      Rendering.Enable(Rendering.Enable_Lighting);

   end Open_Window;

begin -- Simple2

   -- Initialize Visual Simulation Services
   Vis_Sim.Initialize;

   -- Configure multiprocessing mode
   Vis_Sim.Configure;

   -- Read a single FLIGHT-format file and attach geometry to scene
   Scene := Vis_Sim.New_Scene;
   Root  := Import_Utilities.Load_Flt("box.flt");
   Vis_Sim.Add_Child(Scene,Root);
```

```
   -- Configure and open pipeline (defaults to pipe 0)
   Pipe := Vis_Sim.Get_Pipe;
   Vis_Sim.Initialize_Pipe(Pipe,Open_Window'Address);

   -- Get and configure channel
   Channel := Vis_Sim.New_Channel(Pipe);
   Vis_Sim.Attach_Channel_To_Scene(Channel,Scene);
   Vis_Sim.Set_Clipping_Planes(Channel,1.0,1000.0);
   Vis_Sim.Set_Field_Of_View(Channel,45.0,-1.0);

   declare -- Simulate for twenty seconds
      Sine_Value,
      Cosine_Value : Float32;
      Viewpoint    : Math_Utilities.Coordinates_Type;
      Error_Code   : Integer;
   begin
      while Elapsed_Time < 20.0 loop
         -- Sleep until next frame time
         Error_Code := Vis_Sim.Synchronize;

         -- Compute new view position
         Elapsed_Time := Rendering.Get_Time;
         Math_Utilities.SinCos(Float32(33.0*Elapsed_Time),
                               Sine_Value'Address,
                               Cosine_Value'Address);
         Math_Utilities.Set_Vector_3d(Viewpoint.Hpr'Address,
                                      10.0*Sine_Value,
                                     -45.0+10.0*Cosine_Value,
                                      0.0);
         Math_Utilities.Set_Vector_3d(Viewpoint.Xyz'Address,
                                      0.0,
                                     -50.0,
                                      50.0);
         Vis_Sim.Set_Viewpoint(Channel,
                               Viewpoint.Xyz'Address,
                               Viewpoint.Hpr'Address);

         -- Initiate cull/draw for this frame
         Vis_Sim.Frame;
      end loop;
   end;

   -- Clean up and exit
   Vis_Sim.Cleanup;

end Simple2;
```

Figure 46. Simple2, an ObjectSim Low-Level Services version of Simple

Figure 47 shows the Windows package of the LLS. This package exemplifies how Ada renaming declarations are used to mask dependencies on platform-specific libraries. The current implementation of ObjectSim Low-Level Services is comprised of approximately 1500 lines of Ada 95 code, predominantly of the style shown in Figure 47. These 1500

```
with GL;
package ObjectSim.Low_Level_Services.Windows is

   subtype Window_Id_Type is GL.Window_Id;
   subtype Screen_Id_Type is GL.Screen_Id;
   subtype X_Coordinate is GL.X_Screencoord;
   subtype Y_Coordinate is GL.Y_Screencoord;

   function Screen_Select (Number : Screen_Id_Type)
      return Basic_Types.Int32 renames GL.Scrnselect;

   procedure Foreground renames GL.Foreground;

   procedure Set_Position (Lower_Left_X,
                           Upper_Right_X : X_Coordinate;
                           Lower_Left_Y,
                           Upper_Right_Y : Y_Coordinate)
      renames GL.Prefposition;

   procedure Allow_Resizing renames GL.Winconstraints;

   function Open (Name : String) return Window_Id_Type
      renames GL.Winopen;

end ObjectSim.Low_Level_Services.Windows;
```

Figure 47. The ObjectSim 3.0 Low-Level Services Windows Package

lines cover the majority of the Performer library, but only a small portion of GL. Section 5.2.1 discusses the rationale behind which sections are covered and which are not.

Not only can the LLS be accessed by simulation developers, but it also forms the foundation of the ObjectSim 3.0 API. This portion of ObjectSim is discussed in the next section.

4.4 Application Programming Interface

To use Taligent's terminology, the ObjectSim 3.0 API offers a data-driven application framework which is built on top of Low-Level Services. In general, applications which take advantage of the API will also find cause to use the LLS. For example, the LLS includes math routines which cannot be abstracted to a higher level. The goal, however, is to maximize the usefulness of the API, and thereby minimize the necessity of calling Low-

Level Services. This arrangement, depicted in Figure 48, should prove advantageous to ObjectSim maintainers, because it puts **two** layers of abstraction between applications and platform-specific libraries such as Performer. Future maintainers will thus have some flexibility as to the responsibilities and implementation of each layer.

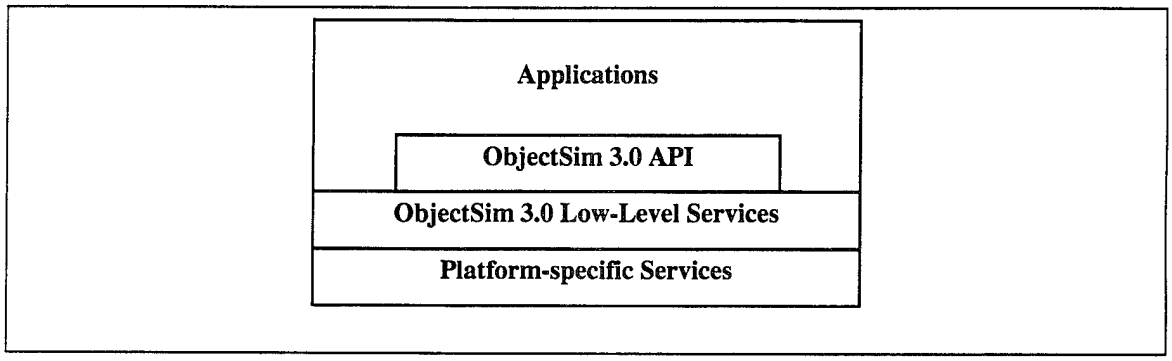

Figure 48. ObjectSim applications will primarily work through the API, but may also access the LLS.

Figure 49 shows excerpts from the Ada 95 package specification for the **View** class of the API. The style of the package is influenced by the Cernosek conventions mentioned in subsection 2.1.2. The class type is declared with the generic identifier **Object**, and the type **Reference** is an access type which designates values of the class type. This setup allows developers to declare instances of the class as **View.Object**, and pointers to instances of the class as **View.Reference**.

Each class has a constructor, **Initialize**, and a destructor, **Finalize**, as well as operations for retrieving and adjusting attribute values. Methods above and beyond these basic operations collectively form the set of highest-level ObjectSim services. An example in the **View** class is the **Track** operation, which causes a view to continually re-orient itself to face an entity.

```
------------------------------------------------------------------------
--     Unit: ObjectSim.API.View
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim API's View class
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Low_Level_Services.Vis_Sim;
with ObjectSim.API.Scene;
with ObjectSim.API.Entity;
with ObjectSim.API.Coordinate_System; use ObjectSim.API.Coordinate_System;
package ObjectSim.API.View is

   type Object is new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   --
   -- Scene operations
   --
   procedure Set_Scene (Instance     : in out Object;
                        Target_Scene : in     Scene.Reference);
   .
   .
   .
   --
   -- Field of view operations
   --
   procedure Set_Horizontal_FOV (Instance  : in out Object;
                                 New_Angle : in     Angle  := -1.0);
   .
   .
   .
   --
   -- Clipping plane operations
   --
   procedure Set_Far_Clipping_Plane_Distance (Instance     : in out Object;
                                              New_Distance : in     Distance);
   .
   .
   .
   --
   -- Position/Orientation operations
   --
   procedure Set_Position (Instance     : in out Object;
                           New_Position : in     Position_Vector);

   procedure Track (Instance      : in out Object;
                    Target_Entity : in     Entity.Reference);
   .
   .
   .
   --
   -- Copy operations
   --
   procedure Mimic (Original : in     Object;
                    Copy     : in out Object);
```

```
   --
   -- Low_Level_Services operations
   --
   function Get_Channel (Instance : Object) return Vis_Sim.Channel_Type;

   --
   -- Exceptions
   --

   -- raised by Move or Set_Position if position adjustment would
   -- move view outside scene boundaries
   Position_Error : exception;
   .
   .
   .
private

   type Object is new Ada.Finalization.Limited_Controlled with
      record
         Channel                      : Vis_Sim.Channel_Type;
         Scene_Ptr                    : Scene.Reference;
         Horizontal_FOV,
         Vertical_FOV                 : Angle                 := 90.0;
         Near_Clipping_Plane_Distance : Distance              := 1.0;
         Far_Clipping_Plane_Distance  : Distance              := 100_000.0;
         Position                     : Position_Vector       := (0.0,0.0,0.0);
         Orientation                  : Orientation_Vector    := (0.0,0.0,0.0);
         Following                    : Entity.Reference      := null;
         Trackee                      : Entity.Reference      := null;
      end record;

end ObjectSim.API.View;
```

Figure 49. A Portion of the ObjectSim API's View Class

A noteworthy aspect of the API is that most classes include some means of extracting a lower-level abstraction from a higher one. For example, in the **View** class there is an operation called **Get_Channel** to retrieve the Low-Level Services channel value from a view instance. Although the goal is to make it unnecessary for developers to work with Low-Level Services in this way, it is recognized that there may be occasions where manipulation at this lower level is required.

A second adaptation of the **Simple** program, designed to use the ObjectSim Application Programming Interface, is shown in Figure 50. It should be noted at this point that the API includes an "extra" package over and above those related to the classes shown in

```
--
-- simpler.adb - a simple performer program
-- converted from simple.adb
--
-- This version of Simple works with the ObjectSim API.
--
with Basic_Types; use Basic_Types;
with ObjectSim.API.Renderer;
with ObjectSim.API.Scene;
with ObjectSim.API.View;
with ObjectSim.API.Coordinate_System;
with ObjectSim.API.Entity.Orientable.ThreeD;
with ObjectSim.Low_Level_Services.Rendering;
with ObjectSim.Low_Level_Services.Math_Utilities;
procedure Simpler is

   use ObjectSim;
   use ObjectSim.API;
   package ThreeD renames Entity.Orientable.ThreeD;

   Elapsed_Time : Float64                := 0.0;

   Scene1       : Scene.Reference;
   View1        : View.Reference;
   Box          : ThreeD.Reference;
   Viewport1    : Renderer.Viewport_Index;

begin -- Simpler

   -- Initialize Renderer
   Renderer.Initialize;

   -- Allocate and initialize objects
   Scene1    := new Scene.Object;
   View1     := new View.Object;
   Box       := new ThreeD.Object;
   Viewport1 := Renderer.New_Viewport;

   -- Load geometry into Box object
   ThreeD.Load(Box.all,"box.flt");

   -- Add Box to Scene1
   Scene.Add(Scene1.all, New_Entity => Entity.Reference(Box));

   -- Configure View1 (vertical FOV defaults to appropriate angle)
   View.Set_Scene(View1.all, Target_Scene => Scene1);
   View.Set_Horizontal_FOV(View1.all,45.0);
   View.Set_Vertical_FOV(View1.all);

   -- Connect View1 to Viewport1
   Renderer.Set_View(Viewport1,View1);

   declare -- Simulate for twenty seconds
      Sine_Value,
      Cosine_Value : Float32;
      Position     : Coordinate_System.Position_Vector;
      Orientation  : Coordinate_System.Orientation_Vector;
   begin
      while Elapsed_Time < 20.0 loop
         -- Sleep until next frame time
         Renderer.Synchronize;

         -- Compute new view position
         Elapsed_Time := Low_Level_Services.Rendering.Get_Time;
```

```
         Low_Level_Services.Math_Utilities.SinCos(Float32(33.0*Elapsed_Time),
                                                  Sine_Value'Address,
                                                  Cosine_Value'Address);
         Orientation := ( 10.0*Sine_Value,
                         -45.0+10.0*Cosine_Value,
                           0.0);
         Position := (  0.0,
                      -50.0,
                       50.0);
         View.Set_Position(View1.all,Position);
         View.Set_Orientation(View1.all,Orientation);

         -- Initiate cull/draw for this frame
         Renderer.Draw_Frames;
      end loop;
   end;

   -- Clean up and exit
   Renderer.Finalize;

end Simpler;
```

Figure 50. Simpler, an ObjectSim 3.0 API version of Simple

Figure 36. This package, called **Coordinate_System**, collects together constants, types and operations which relate to ObjectSim's world coordinate system. (The Framework version of this package, identical to its API counterpart, is given in Appendix B.)

At about 85 lines, this version of the **Simple** program is indeed simpler than its predecessors. The primary simplification is that the **Renderer** is now responsible for opening the window as part of its initialization. Furthermore, object-oriented programmers will likely find `ThreeD.Load(Box.all,"box.flt")` more natural than `Root := Import_Utilities.Load_Flt("box.flt").`

The final section of this chapter describes how the ObjectSim API was transformed into the ObjectSim Framework. As discussed previously, the API and Framework are quite similar. However, the fundamental difference between the two leads to substantially different application development processes.

4.5 The Application Framework

The differences between ObjectSim's API and Framework were outlined at the end of chapter 3. In terms of source code adjustments, the following simple steps were sufficient to transform the Ada 95 API into the Ada 95 Framework:

- The API source code was copied, and all references to "API" in the copy were changed to "Framework."
- The class types were changed to abstract types because the Framework classes are abstract.
- An abstract **Update** operation was added to the **Scene**, **View**, **Entity** and **Environment** classes.
- A **Scene_Manager** class and a **View_Manager** class were added to the Framework. These classes are concrete.
- An abstract **Simulation** class was added to the Framework. This class is the template for all ObjectSim simulations which use the Framework.

The Ada 95 source code for the package specifications which comprise the ObjectSim 3.0 Application Framework is attached as Appendix B. It may be helpful for the reader to examine this appendix now, since the following subsection assumes an understanding of this code.

4.5.1 Tailoring the Framework

It is easiest to explain the process by which developers tailor the ObjectSim Framework by taking one last look at the **Simple** example used throughout this chapter. A good place to start is with the only entity in this simulation--the box. The box never moves in the course of **Simple**, so the behavior of this entity is easy to describe. Figure 51 shows how the **Box** subclass is a specialized version of the abstract **3D** class.

```
------------------------------------------------------------------------
--     Unit: Box
--   Author: Capt Shawn Hannan
--
-- Comments: a stationary multi-colored cube
--
--  History: 23 Sep 95 - created
--
------------------------------------------------------------------------
with ObjectSim.Framework.Entity.Orientable.ThreeD;
package Box is

   type Object is new ObjectSim.Framework.Entity.Orientable.ThreeD.Object
      with private;

   type Reference is access all Object'Class;

   procedure Update (Instance : in out Object);

private

   type Object is new ObjectSim.Framework.Entity.Orientable.ThreeD.Object
      with null record;

end Box;

------------------------------------------------------------------------
package body Box is

   procedure Update (Instance : in out Object) is
   begin
      null;
   end Update;

end Box;
```

Figure 51. The Box subclass is derived from the Framework's 3D class

Slightly more complicated is the rotating view which gives motion to the **Simple** simulation. Figure 52 shows how the **Rotating_View** subclass is derived from the Framework's **View** class, and how the behavior of the subclass is defined in the **Update** operation.

All that remains is to define a specialized subclass of the **Scene** class. In the case of **Simple**, the entire scene (comprised of just the one box) is stationary, so the **Update** operation need not do anything. In general, however, **Scene** subclasses will want to

```
------------------------------------------------------------------------
--     Unit: Rotating_View
--   Author: Capt Shawn Hannan
--
-- Comments: the rotating view of the Simplest simulation
--
--  History: 23 Sep 95 - created
--
------------------------------------------------------------------------
with ObjectSim.Framework.View;
package Rotating_View is

   type Object is new ObjectSim.Framework.View.Object with private;

   type Reference is access all Object'Class;

   procedure Update (Instance : in out Object);

private

   type Object is new ObjectSim.Framework.View.Object with
      null record;

end Rotating_View;


------------------------------------------------------------------------
with Basic_Types; use Basic_Types;
with ObjectSim.Low_Level_Services.Rendering;
package body Rotating_View is

   use ObjectSim;

   procedure Update (Instance : in out Object) is

      Sine_Value,
      Cosine_Value : Float32;

      Position     : Framework.Coordinate_System.Position_Vector;
      Orientation  : Framework.Coordinate_System.Orientation_Vector;

      Elapsed_Time : Float64;

   begin

      Elapsed_Time := Low_Level_Services.Rendering.Get_Time;
      Low_Level_Services.Math_Utilities.SinCos(Float32(33.0*Elapsed_Time),
                                               Sine_Value'Address,
                                               Cosine_Value'Address);
      Orientation := ( 10.0*Sine_Value,
                      -45.0+10.0*Cosine_Value,
                        0.0);
      Position := (  0.0,
                   -50.0,
                    50.0);
      Set_Position(Instance,Position);
      Set_Orientation(Instance,Orientation);

   end Update;

end Rotating_View;
```

Figure 52. The Rotating_View subclass is derived from the Framework's View class

update themselves by calling **Update** for all entities in the scene, as well as for the environment, if there is one. The **Simple_Scene** subclass, shown in Figure 53, demonstrates this process.

```
------------------------------------------------------------------------
--     Unit: Simple_Scene
--   Author: Capt Shawn Hannan
--
-- Comments: the scene of the Simplest demo
--
--  History: 23 Sep 95 - created
--
------------------------------------------------------------------------
with ObjectSim.Framework.Scene;
package Simple_Scene is

   type Object is new ObjectSim.Framework.Scene.Object with private;

   type Reference is access all Object'Class;

   procedure Update (Instance : in out Object);

private

   type Object is new ObjectSim.Framework.Scene.Object with
      null record;

end Simple_Scene;

------------------------------------------------------------------------
with ObjectSim.Framework.Entity;
package body Simple_Scene is

   use ObjectSim.Framework;

   procedure Update (Instance : in out Object) is
   begin

      -- This can be a bit confusing.  Because Simple_Scene is not a
      -- child of the Framework's Scene class, it does NOT have access
      -- to the private components of its own record!  That means we
      -- cannot reference Instance.Number_Of_Entities, or the set
      -- Instance.Entities, and so on.  We therefore use public
      -- functions to access this information.

      for Index in 1..Scene.Number_Of_Entities_In(Scene.Object(Instance)) loop
         Entity.Update(Scene.Get_Entity(Scene.Object(Instance),Index).all);
      end loop;

   end Update;

end Simple_Scene;
```

Figure 53. The Simple_Scene subclass is derived from the Framework's Scene class

Given the **Box**, **Rotating_View** and **Simple_Scene** classes, the next step is to write the program which brings them all together. This is accomplished by creating a new subclass of the Framework's **Simulation** class, which specifies a relatively simple structure for all applications based on the Framework. The structure breaks applications into three parts: initialization, visualization, and finalization. The **Simulation** class includes a method for each of these three simulation phases. Each method has default behavior, described below:

- **Initialize** - initializes the **Renderer**, **Scene_Manager** and **View_Manager**
- **Finalize** - finalizes the **Renderer** (the managers are finalized by the run-time system)
- **Visualize** - infinitely loops, calling **Update** for the **Scene_Manager** and **Update** for the **View_Manager** during each iteration

Application developers will need to override the **Initialize** method to set up their scenes as desired. In many cases, the default behavior for **Finalize** and **Visualize** will be sufficient, and will not be overridden. Developers must exercise caution to replicate the required behavior of overridden methods. For example, when **Initialize** is overridden, the developer must ensure the **Renderer** and manager instances are initialized. This is done easily enough by calling the **Simulation** superclass version of **Initialize** as part of the implementation of the subclass method. Figure 54, which shows **Simplest**, the Framework version of the **Simple** simulation, demonstrates this technique.

```
--
-- simplest.ads - a simple performer program
--
with ObjectSim.Framework.Renderer;
with ObjectSim.Framework.Scene;
with ObjectSim.Framework.View;
with ObjectSim.Framework.Entity.Orientable.ThreeD;
with ObjectSim.Framework.Simulation;
package Simplest is

   use ObjectSim.Framework;
   package ThreeD renames Entity.Orientable.ThreeD;

   type Object is new Simulation.Object with private;

   type Reference is access all Object'Class;

   procedure Initialize (Instance : in out Object);
   -- Visualize and Finalize accept default behavior

private
   type Object is new Simulation.Object with
      record
         Scene1    : Scene.Reference;
         View1     : View.Reference;
         Viewport1 : Renderer.Viewport_Index;
         Box1      : ThreeD.Reference;
      end record;
end Simplest;
-------------------------------------------------------------------------
with Box;
with Rotating_View;
with Simple_Scene;
with ObjectSim.Framework.Scene_Manager;
with ObjectSim.Framework.View_Manager;
package body Simplest is

   procedure Initialize (Instance : in out Object) is
   begin
      -- As required, call superclass Initialize
      Simulation.Initialize(Simulation.Object(Instance));

      -- Allocate and initialize objects
      Instance.Scene1    := new Simple_Scene.Object;
      Instance.View1     := new Rotating_View.Object;
      Instance.Box1      := new Box.Object;
      Instance.Viewport1 := Renderer.New_Viewport;

      -- Load geometry into Box object
      ThreeD.Load(Instance.Box1.all,"box.flt");

      -- Add Box to Scene1
      Scene.Add(Instance.Scene1.all,
                New_Entity => Entity.Reference(Instance.Box1));

      -- Configure View1 (vertical FOV defaults to appropriate angle)
      View.Set_Scene(Instance.View1.all,
                             Target_Scene => Instance.Scene1);
      View.Set_Horizontal_FOV(Instance.View1.all,45.0);
      View.Set_Vertical_FOV(Instance.View1.all);

      -- Connect View1 to Viewport1
      Renderer.Set_View(Instance.Viewport1,Instance.View1);

      -- Inform managers
      Scene_Manager.Add(Simulation.Scene_Manager_Of(Simulation.Object(Instance)).all,
                        Instance.Scene1);
      View_Manager.Add(Simulation.View_Manager_Of(Simulation.Object(Instance)).all,
                       Instance.View1);
   end Initialize;
end Simplest;
```

Figure 54. Simplest, the Framework version of the Simple simulation

Figure 55 depicts the **Simplest** simulation. Note the similarity to the ObjectSim 3.0 Application Framework diagram of Figure 44.

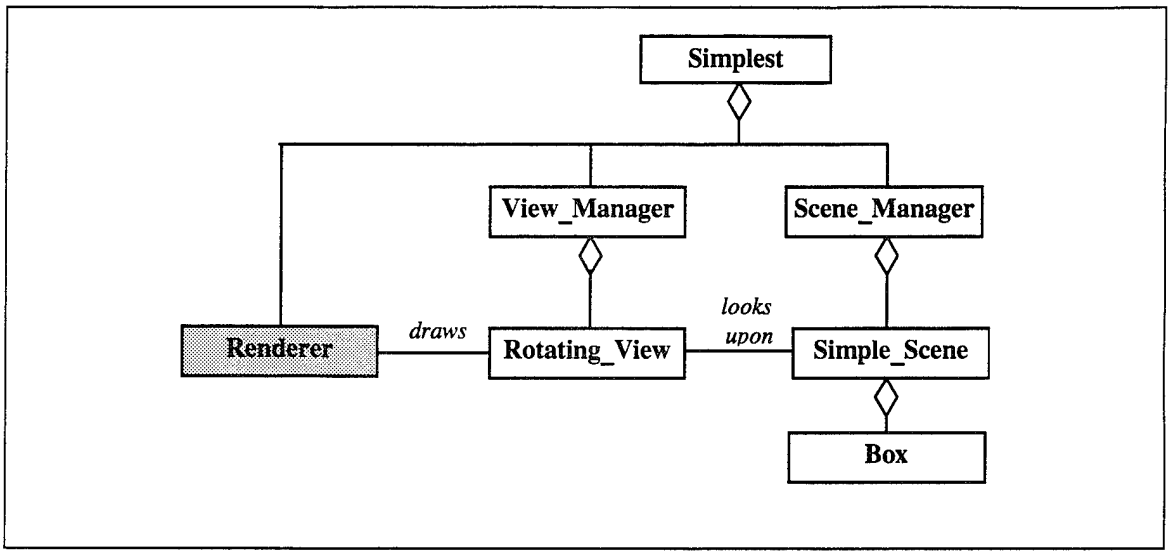

Figure 55. Simplest is a specialized version of the Application Framework

With the specialization of the Framework complete, the final step is to write a short procedure to "kick off" **Simplest**. This procedure is shown in Figure 56.

```
--
-- simplest_driver.adb
--
-- Initialize is called automatically when S is elaborated.  Visualize
-- is called explicitly as the only executable line of this driver.
-- Finalize is called automatically after Visualize is complete, but
-- before Simplest_Driver terminates.
--
with Simplest;
procedure Simplest_Driver is
  S : Simplest.Reference := new Simplest.Object;
begin
  Simplest.Visualize(S.all);
end Simplest_Driver;
```

Figure 56. Simplest_Driver executes the simulation

4.5.2 Framework Summary

The case could likely be made that **Simplest** is actually the **least** simple version of the **Simple** program. Whether using Performer and GL directly, working through the ObjectSim Low-Level Services layer, or taking advantage of the ObjectSim API, all prior versions of the application were written as a single procedure. **Simplest**, on the other hand, was created by writing four Ada packages and a short driver routine. In terms of coding, then, **Simplest** is more complex than its predecessors, if one accepts the argument that more components implies greater complexity.

The counterargument is that modularization is a means of **dealing with** complexity. Since the ObjectSim Framework standardizes the modularization of simulation applications, the inherent complexity of those applications is always dealt with in the same way. The use of a recurring design should allow developers to create and maintain applications more easily than if every application were uniquely structured. Chapter 5 discusses this issue further.

5. Results and Comparisons

5.1 Overview

The goal of this chapter is to scrutinize the latest version of ObjectSim, both in isolation and in comparison to its predecessor, Easy_Sim. Section 5.2 takes a closer look at the three parts of ObjectSim 3.0, identifying the strengths and weaknesses of those parts. Section 5.3 examines improvements and shortcomings of ObjectSim 3.0 compared to Easy_Sim.

5.2 An Analysis of ObjectSim 3.0

The next three subsections offer an analysis of ObjectSim's Low-Level Services, Application Programming Interface and Application Framework. The intention is to identify results which contribute to the "bottom line" of what ObjectSim 3.0 has achieved.

5.2.1 Low-Level Services

The purpose of the LLS is to keep applications from relying on platform-specific libraries without denying applications the capabilities of those libraries. If future Lab applications are implemented using the LLS, the potential problem of porting those applications to new platforms is largely reduced to the problem of porting the LLS. Though moving ObjectSim's Low-Level Services to a non-SGI platform would likely prove an arduous task, the isolation of platform-dependencies in this layer of the architecture still represents a substantial improvement over the Lab's status quo.

Unfortunately, there are three foreseeable drawbacks to having new applications call the LLS in lieu of calling the IRIS Graphics Library and Performer. First is the fact that the LLS, as presently implemented, does not cover the full spectrum of GL and Performer capabilities. With respect to GL, the LLS currently provides only a small subset of SGI's routines. This is due to the fact that, in general, visual simulations need only call GL during initialization to open and configure the graphics window. Thus the initial implementation of Low-Level Services offers just the few GL-related routines needed for this purpose. At this point, any applications requiring additional services from GL would have no choice but to call GL itself. Since existing applications do, in fact, take advantage of GL for more than opening the graphics window, it is clear that the relationship between Low-Level Services and GL must be examined in the future. (See subsections 6.2.2.2 and 6.2.2.3.)

With respect to Performer, the LLS includes counterparts to all constants, types and subprograms which appear in the Ada-to-C Performer bindings supplied by SGI. Regrettably, these bindings are not complete. That is, there are Performer routines which can be called from C programs, but which are presently inaccessible from Ada programs. Since the LLS relies on the bindings, there are no LLS counterparts for any Performer entities which are missing from the bindings. Fortunately, the omissions are relatively minor. During implementation of the ObjectSim 3.0 demonstration programs, all required Performer capabilities were found to be available. Furthermore, once up-to-date,

complete bindings are obtained from Silicon Graphics, it should take minimal effort to incorporate the new features into the LLS.

A second concern with using the LLS in place of direct calls to GL and Performer relates to programming languages. ObjectSim 3.0 has only been implemented in Ada 95, but Lab simulations have historically been developed in C and C++. Either the LLS must be translated to C or C++, or applications which use it must be written in Ada 95.

The third LLS issue is documentation. In the graphics sequence at AFIT, students learn how to use SGI's GL and Performer libraries. Though the correlation between Low-Level Services and these libraries is extremely close, the names of all constants, types and subprograms are different. For the most part, the LLS identifiers are less cryptic (e.g., "Attach_Channel_to_Scene" versus "pfChanScene"), but the differing identifiers nevertheless represents a difficulty for developers. Appendix C contains a set of cross-reference tables which list the LLS routines and their SGI counterparts. This appendix represents the only current documentation of Low-Level Services.

5.2.2 The API versus the Framework

Chapter 4 showed how the **Simple** program was implemented using in one case the ObjectSim 3.0 API, and in another case the ObjectSim 3.0 Framework. Implementation of the API version of the program was straightforward. The main procedure simply declared one instance of the **Scene** class, one instance of the **View** class and one instance of the **3D**

Entity class. By manipulating these instances, as well as making key calls to the **Renderer**, the **Simple** simulation was achieved with about 85 lines of Ada source code.

The Framework version of **Simple** was relatively complex by comparison. Specialized subclasses of **Scene, View** and **3D**--as well as the Framework **Simulation** class--had to be written. The former three subclasses included a concrete **Update** operation which specified the behavior of subclass instances, while the latter subclass had a specialized **Initialize** procedure. The overall simulation was further complicated by two additional class instances, a **Scene_Manager** instance and a **View_Manager** instance. The Framework version also used the **Renderer** object.

Simply put, it is easier to understand and use the API than the Framework. This does not imply, however, that it is **better** to use the API than the Framework. Whereas API-based simulations will merely share a collection of related classes, simulations based on the Framework will share a certain basic structure. Once this basic structure is understood, future maintainers of Framework applications will be able to appreciate the common foundation which underlies those applications.

Furthermore, since the ObjectSim Framework dictates the structure of conforming applications, developers are relieved of the burden of designing that structure from scratch. Hopefully, then, new simulation developers will get their applications off the

ground faster than without the Framework, once over the initial hurdle of learning the base design.

5.2.3 A Closer Look at the Framework

Although using the ObjectSim Framework requires more developer savvy than using the API, the Framework is actually conceptually fairly simple. In fact, its complexity is comparable to that of Easy_Sim. The following two subsections examine in more detail what the Framework does, and does not do, for developers.

5.2.3.1 What the Framework Does

The ObjectSim Application Framework establishes a basic design for visual simulations. This design will be tailored, via inheritance, for each application. Any application based on the Framework will have certain characteristics. First, an application will be represented by a new subclass of **Simulation**, and will be divided into three phases: initialization, visualization and finalization. The initialization phase will be unique to each simulation, though every simulation must call the superclass version of **Initialize**, because this routine performs mandatory startup functions. The visualization phase defaults to an endless loop, during which the **Scene_Manager** and **View_Manager** are instructed to update the scenes and views, and the **Renderer** is instructed to draw the views to the screen. The finalization phase performs any necessary post-simulation cleanup. The default behavior of the latter two phases will likely be sufficient for many applications.

Second, as far as composition, an application will probably include exactly one instance of a **Scene** subclass, exactly one instance of an **Environment** subclass, at least one instance of a **View** subclass and multiple instances of a variety of **Entity** subclasses.

Third, the behavior of every **Scene**, **View**, **Entity** and **Environment** instance is dictated by the **Update** operation for the instance's class. For each frame of the simulation, the managers will call **Update** for each scene and view. It is up to developers, however, to ensure the environment (if there is one) and all entities are updated. This should be done as part of the **Update** routine for each **Scene** subclass. (See Figure 53 for an example.)

5.2.3.2 What the Framework Does Not Do

There are two important things the Framework does not do for developers. First, the Framework does not assist with dealing with user input. Although this may at first seem to be a glaring oversight, input-handling was omitted only after careful consideration. Given that there are so many kinds of input (keyboard, mouse, head-mounted display, joystick, etc.) and so many purposes for input (executive control, entity manipulation, view manipulation, etc.), there was no clear way to make input-handling part of the Framework. It was therefore decided to leave input-handling to application developers as part of the tailoring process.

The second thing the Framework does not handle for developers is interrelated object behavior. The **Update** operation of every subclass is independent of the **Update**

operation for every other subclass. For example, consider a simulation with two views, **View1** and **View2**. Assume the **Update** operation for **View1** specifies the view will maintain a certain location, but will rotate about its Z-axis. Now assume the second view is supposed to remain stationary until the first view rotates to a certain orientation, then the second view is to begin rotating as well. This means **View2** must have access to **View1**'s orientation. The only way this can happen is for the developer to write a special routine for **View2** which allows it to receive a pointer to **View1**, since no such operation is inherited from the basic **View** class. Indeed, except for the relationships inherent to the ObjectSim Framework, no class instances are aware of any other class instances, unless the developer adds routines for passing such information.

Since it would be impossible for the Framework to predict which instance behaviors are related to which other instance behaviors, it is no surprise that this sort of responsibility falls to the application developers. Still, it is important to recognize that the Framework divides the overall behavior of simulations into well-defined partitions, but does not assist in any interactions among those partitions.

5.3 ObjectSim 3.0 Versus Easy_Sim

The objective of this section is to discuss key differences between the ObjectSim 3.0 architecture and its predecessor architecture, Easy_Sim. The following subsections examine these differences, beginning with ObjectSim's improvements versus Easy_Sim, and ending with what might be deemed comparative weaknesses of the new architecture.

5.3.1 The Scene Class

Although the Clip and Vega APIs discussed at the beginning of chapter 3 are substantially different visual simulation products, they share many fundamental concepts. Among these is the notion that a *scene*, or *world* in the case of Clip, is the heart of a visual simulation. The scene or world is the virtual environment in which the simulation takes place. Although there are a myriad of other contributing classes, the class representing the collective scene is central to the abstraction of a visual simulation.

Interestingly, neither the original ObjectSim architecture nor its successor Easy_Sim include a scene class as part of the design. In both architectures, the abstraction of the scene is considered a part of the **View** class. At the start of an Easy_Sim simulation, for example, an instance of the **View** class is created, which implicitly creates the overall scene. Subsequently, a player can be added to the simulation via a routine in the **Simulation** class which effectively says, "Add this player to the scene implicitly associated with this view." The specification of this routine is:

```
procedure Add (Instance    : in out Simulation.Object;
               New_Player  : in     Player.Reference;
               To_View     : in     View.Reference;
               Model_File  : in     String;
               With_Coords : in     Coords;
               Coords_File : in     String);
```

This is less intuitive than the ObjectSim 3.0 approach, in which a routine in the **Scene** class is used. This routine says simply, "Add this entity to this scene." The specification of this routine is:

```
procedure Add (Instance   : in out Scene.Object;
               New_Entity : in     Entity.Reference);
```

And because with ObjectSim 3.0 there are two distinct classes with an explicit *look upon* association between them (Figure 44), it is possible to have a view switch from one scene to another in a multi-scene simulation. This isn't possible with Easy_Sim, because the scene is fundamentally a part of the view.

Furthermore, this issue casts a shadow of doubt on Easy_Sim's capability to deal with multiple views of a single scene. When an instance of the Easy_Sim **View** class is created, a new scene is automatically created as well. It is unclear how two views can be associated with the same scene. Kayloe's thesis states Easy_Sim has this capability, but admits it was never attempted [Kay94, 98-101].

5.3.2 Entity versus Player

Easy_Sim's **Player** class includes attribute values for both position and orientation, as well as operations for setting and retrieving these values. In general, this approach makes sense, since most players will have a dynamic position and orientation within the scene. The exception is in the case of a player which has no orientation, such as an omni-directional light source. With Easy_Sim, a **Light_Source** subclass of **Player** could be designed with new attributes for, say, color and intensity. The superclass orientation attributes would simply be ignored for instances of the **Light_Source** subclass.

ObjectSim 3.0 exchanges the **Player** class for the more generically-termed **Entity** class, recognizing that not all entities in a simulation are orientable, positionable "players." The

abstract **Entity** class has **no** attributes, merely serving as the root of the class hierarchy of all things which could conceivably be placed in a scene, as shown in Figure 57. (The ObjectSim approach **does** assume that all entities have a position in the scene, but position information is maintained in the **Scene** class. This decision will be discussed further in subsection 5.3.8.)

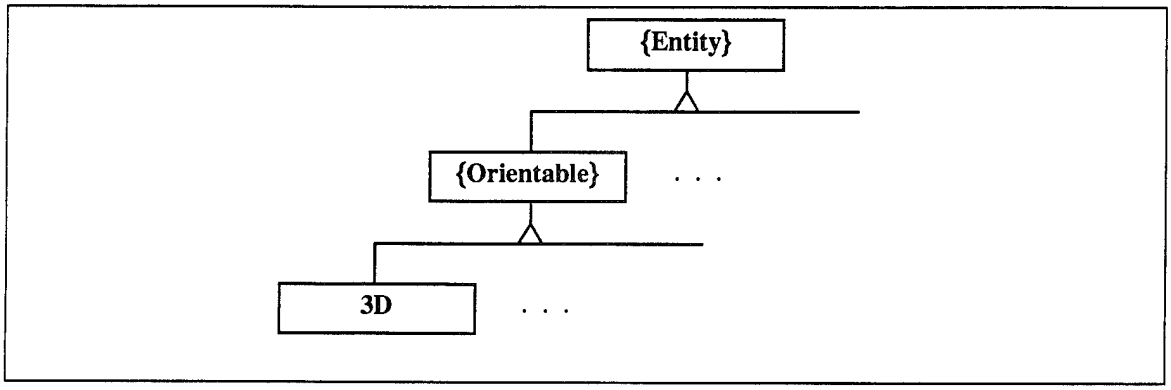

Figure 57. The ObjectSim 3.0 Framework's Entity Hierarchy

In short, the ObjectSim 3.0 architecture partially specifies an entire hierarchy of simulation entities, whereas Easy_Sim offers only the **Player** class. Future expansion of this hierarchy would likely result in an arrangement similar to that shown in Figure 58.

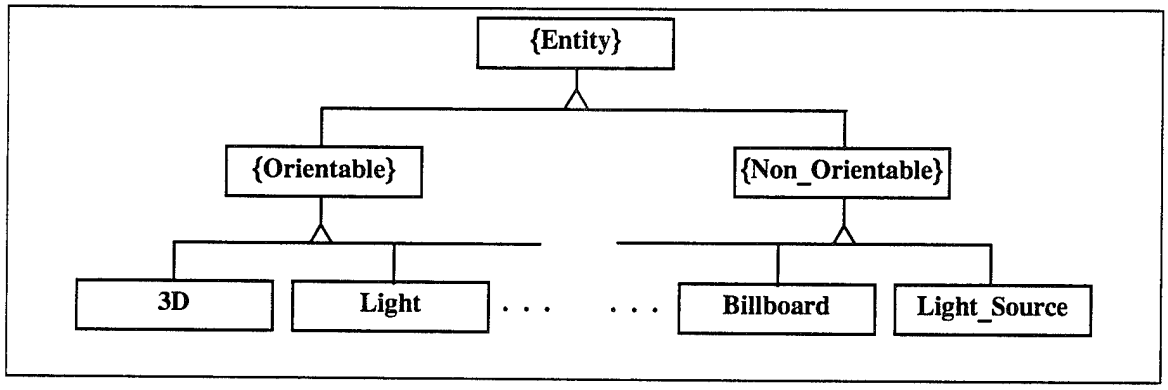

Figure 58. Possible expansion of Entity Hierarchy

Foreseeable non-orientable classes include not only the **Light_Source** class discussed above, but also a **Billboard** class. (See [Har94, 115] for an explanation of billboards.) A likely new orientable subclass would be a **Light** class for lights which shine in a certain direction.

5.3.3 Entities, Players and Models

The Easy_Sim architecture has both a **Player** class and a **Model** class. In general, an instance of the **Player** class has some associated geometry which appears on-screen. This geometry is stored as an instance of the **Model** class. Thus, in an Easy_Sim simulation, the developer is responsible for maintaining two class instances for each visible player--the **Player** instance, and the associated **Model** instance. Although not stated explicitly in Kayloe's thesis, the reason for having two distinct classes relates to efficiency. It is possible for many players to have the same visible geometry. There may therefore be a many-to-one relationship from **Player** instances to a single **Model** instance. By offering two separate classes, Easy_Sim gives the developer control of which players are associated with which models. This design also allows players to alter their geometry in the midst of the simulation.

The ObjectSim 3.0 architecture combines the aforementioned classes into a single **3D Entity** class. The geometric model is an inherent part of this class, so developers are not burdened with maintaining two separate class instances for each entity. This conceptual simplification currently causes a potential efficiency loss. ObjectSim does not presently allow model-sharing by multiple entities, though it would be quite simple to add a

Share_Model routine to the **3D** class. Nor does ObjectSim presently allow **3D** instances to change models during the simulation, other than by loading new geometry from a file, which would likely be prohibitively slow. This could be overcome by declaring multiple instances of the **3D** class for a single entity, then loading the geometry for each instance from a different model file. During the simulation, the various instances could be swapped into or out of the scene, depending on which geometric representation was desired. This solution seems at least as burdensome on the developer as the Easy_Sim method, but the expense of this special case is offset by the complexity savings in the more general case where an entity has a single associated model, as described above.

5.3.4 Multiple Views

Subsection 5.3.1 pointed out the potential difficulty of working with multiple views in Easy_Sim. This difficulty is compounded by the fact that Easy_Sim does not give developers the capability to manipulate viewports, which makes it unclear how Easy_Sim simulations can have multiple views drawn to a single screen simultaneously.

ObjectSim 3.0 allows the screen to be partitioned into several viewports via calls to the **Renderer**. Each instance of the **View** class can be "attached to" one or more viewports. These "attachments" can be changed mid-simulation, giving the developer full control over which views are displayed where at all times.

5.3.5 Following and Tracking

The Easy_Sim **Player** class includes an operation called **Look_At**, which reorients a player such that it faces a particular point. This same operation has been carried over to ObjectSim 3.0, for both entities and views. Furthermore, ObjectSim has new capabilities not found in Easy_Sim. As discussed in subsection 3.4.3.2, both entities and views can be instructed to follow a "leader" entity. As the leader moves, all followers are automatically moved by the Framework. Secondly, ObjectSim's **View** class has a **Track** operation, which causes a view to constantly reorient itself to face a selected entity. Again, this is accomplished automatically by the Framework during each frame of the simulation.

5.3.6 User Input

Subsection 5.2.3.2 discussed the fact that the ObjectSim 3.0 Framework has no provisions for dealing with user input. Although, as mentioned in that subsection, this was a conscious decision, it must be noted that this decision caused a slight capability loss versus Easy_Sim. The Easy_Sim architecture includes a **Modifier** class, where each instance of this class is associated with an instance of the **View** class. A modifier, such as a keyboard, can be used to change the position and orientation of an associated view. However, by explicitly offering this capability, Easy_Sim draws attention to the fact that it has no provisions for handling input for other purposes, such as player movement or executive control (e.g., stop the simulation immediately). So although ObjectSim has no equivalent to Easy_Sim's **Modifier** class and **Modifier-View** association, it is more consistent in the

sense that user input of all kinds, for any purpose, is the responsibility of the tailorer of the Framework.

5.3.7 Environment

Easy_Sim and ObjectSim differ substantially in their views of a scene's environment. With Easy_Sim, the **Environment** class has a mandatory association with the **Model** class. This means that every **Environment** instance must include a geometric representation, such as a mountainscape or city block. While this is convenient for many types of simulations, it is unnecessarily restrictive; some simulations may require a simple background color, with no associated 3D model.

ObjectSim 3.0, by contrast, does not associate a 3D model with the environment. Instead, the **Environment** class represents almost exactly the same abstraction as the Performer pfEarthSky. (See [Har94, 185] for a discussion of the pfEarthSky.) This means the **Environment** instance may establish things such as a solid background color, or perhaps a sky of varying shades combined with a terrain of varying shades. In no case, however, would an ObjectSim environment include buildings, mountains, or any other three-dimensional objects. These sorts of things would be classified as entities. This raises another issue, which is discussed in the next subsection.

5.3.8 Static versus Dynamic Entities

ObjectSim 3.0 does not distinguish between static and dynamic entities. This could potentially impact the speed of the simulation, depending on the implementation of the

Framework. Since it is usually more computationally expensive to render frames with moving entities than it is to render frames with stationary entities, the speed of ObjectSim 3.0 simulations may suffer, because all entities are considered moveable. This simplification releases developers from the task of partitioning simulation entities into two groups, static and dynamic. The run-time cost of this simplification, however, necessitates further consideration of this issue. (See subsection 6.2.1.4.)

5.3.9 The Division of Labor

A long-recognized principle of software engineering is that of *modularity*, which asserts that the complexity of a program should be divided into manageable partitions. It is undesirable to have a large, monolithic program in which all functionality is contained in a single module. A program which has been decomposed into a collection of cooperating components is easier to understand and maintain.

Arguably, the ObjectSim 3.0 Framework is more monolithic than its Easy_Sim counterpart. As pointed out in chapter 3, ObjectSim's **Scene** class is the most elaborate of the Framework, in the sense that it has the richest set of operations. This is due in large part to the "simplicity principle" discussed in subsection 3.4.3.2, which mandates a one-way hierarchy of class dependencies, as shown in Figure 37. This hierarchy establishes that an entity is not aware that it is positioned within a scene, or that a scene even exists. Therefore, an entity's scene position, as well as operations for moving the entity within the scene, are part of the **Scene** class. These include the **Move**, **Set_Position**, **Follow** and

Stop_Follow routines, as well as the functions which return entity position information. The end result is that many entity functions may appear to be misplaced in the **Scene** class, while the **Entity** class has no operations of its own, other than the abstract **Update** procedure. In this sense, the "division of labor" within ObjectSim may seem top-heavy, as evidenced by the Rumbaugh diagram of Figure 59.

In an Easy_Sim simulation, the **Player** class shoulders more of the overall workload than the **Entity** hierarchy does in an ObjectSim 3.0 simulation. The case could be made that ObjectSim's **Entity** class should be "strengthened" to be more like a **Player** by including the scene position information as a class attribute, and by moving position-related operations from **Scene** to **Entity**. This would better balance the division of labor in an ObjectSim simulation, at the cost of losing the one-way class dependency hierarchy: **Scene** instances would still need to call the **Update** operation for each entity, and **Entity** instances would need to call the **Scene** class to ensure their positions were within the scene boundaries.

Such an adjustment would have an additional benefit, but this advantage will be discussed in chapter 6.

5.3.10 Summary

ObjectSim 3.0 differs from Easy_Sim both in concept and in capability. Conceptually, key differences include a new **Scene** class, an **Entity** class hierarchy in lieu of Easy_Sim's **Player** and **Model** classes, and a different abstraction of "the environment."

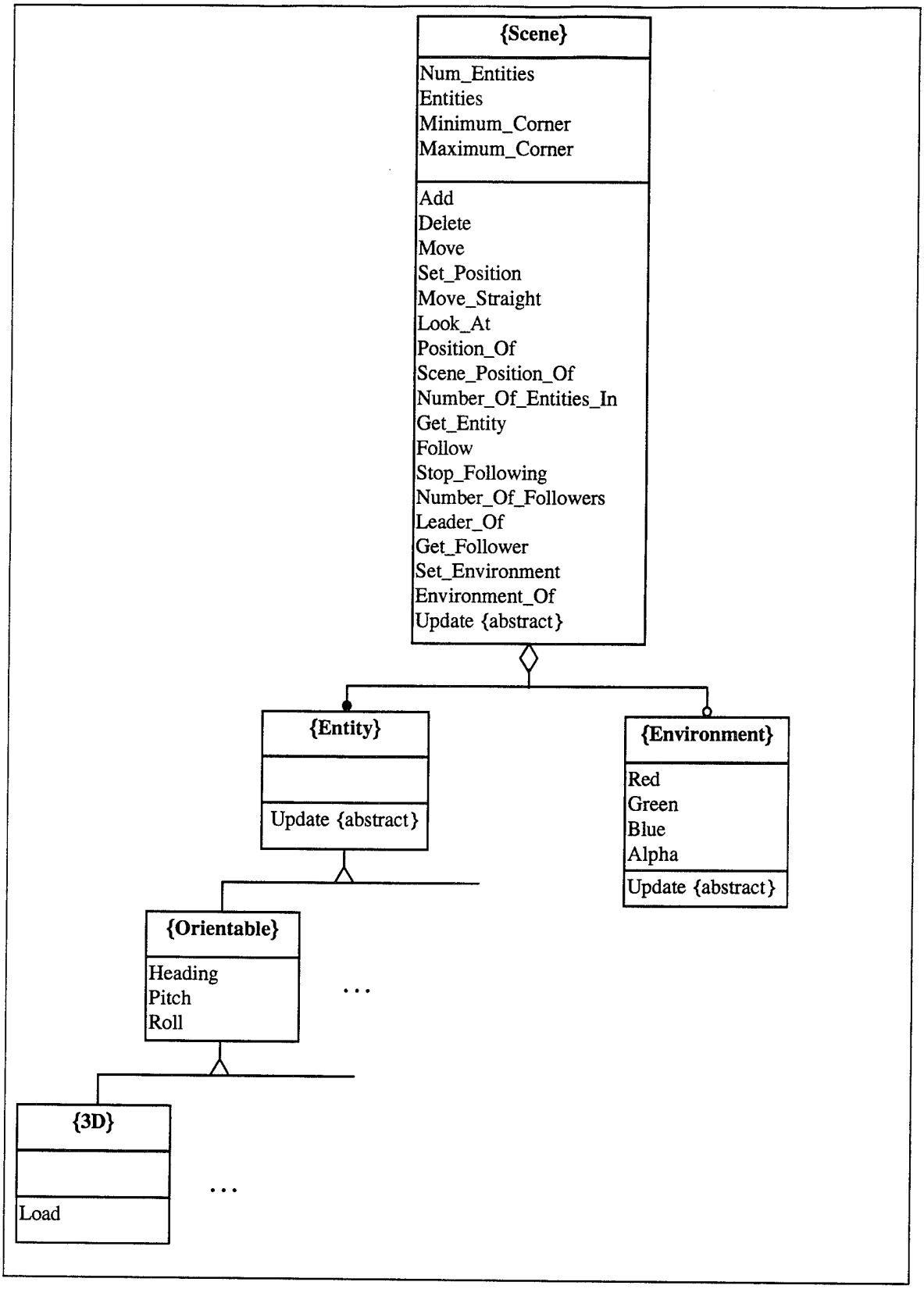

Figure 59. The Scene class and its components

In terms of capability, ObjectSim adds to its predecessor the ability to set up multiple viewports and dynamically attach views to those ports. The new architecture also includes "following" and "tracking" features not found in Easy_Sim. Lastly, ObjectSim leaves all aspects of input-handling to simulation developers, whereas Easy_Sim assists developers in dealing with input for view adjustment, but nothing else.

6. Recommendations for Future Study

6.1 Overview

The objective of this final chapter is primarily to offer suggestions for continued work in this research area. Much of the discussion of recommendations in section 6.2 focuses on the results and issues described in the preceding chapter. Section 6.3 closes out the document with final remarks.

6.2 Recommendations

The recommendations for future study are broken down into two categories: technical improvements and strategic recommendations. The technical improvements subsection proposes adjustments to the ObjectSim 3.0 architecture as it now stands, and lists the benefits of those proposals. The strategic recommendations subsection concerns suggestions for long-term planning in the visual simulation software architecture research area, based on experiences during this thesis effort.

6.2.1 Technical Improvements

6.2.1.1 Remove Renderer

The **Renderer** is an anomaly in the ObjectSim Framework. It is a standalone object rather than a class. It was designed largely as the collecting place for routines which did not fit in other classes. The functionality of the **Renderer** could be distributed to other classes, namely **Simulation** and **View_Manager**. Executive control operations, such as setting or examining the frame rate, could go to **Simulation**, while routines related to the viewports could be moved to **View_Manager**. Another possibility would be to add a **Viewport**

class, some sort of **Display** class, or both. Adding new classes would be the most natural, object-oriented way to parcel out this functionality, at the cost of greater architectural and developmental complexity.

6.2.1.2 Change the Scene/Entity Division of Labor

As discussed in Chapter 5, the **Scene** class probably does too much, and the **Entity** class probably does too little. The classes would be better balanced if the **Entity** class included position attributes, as well as all operations related to position. This would make implementation harder, because **Entity** and **Scene** would be mutually-dependent classes.

The end of Chapter 5 alluded to a potential benefit beyond an improved division of labor. Given the suggested changes, a new Framework **Follower** subclass of **Entity** could be designed. This would be a special kind of entity which is **always** following some other entity. The concept could be expanded to other kinds of specialized entities. A **Mimic** might follow a lead entity's movements, and also copy any orientation changes. A **Shadow** might maintain a relative position in world coordinates, whereas the **Follower** would maintain a relative position in the coordinate system of the leader. In short, a variety of **Entity** subclasses could be designed, each with a particular behavior related to position and orientation.

6.2.1.3 New Kinds of Views

Similar to the previous proposal, new subclasses of the Framework's **View** class could be designed as well. There could be **Follower**, **Mimic** and **Shadow** views. There could also

be a **Tracker**, which would be a view which is **always** tracking some entity. A **Tracker_Follower** could track one entity while following another.

6.2.1.4 Static Entities

ObjectSim does not presently distinguish between static and dynamic entities. This can have important consequences at run-time. As long as there is the potential that an entity will move, the position and orientation of that entity must be calculated every frame of the simulation. If it is known for certain that the entity will **not** move, these calculations can be performed only once. For the sake of efficiency, ObjectSim should allow developers to specify whether an entity is static or dynamic. One way to achieve this distinction would be with inheritance. The **Entity** class would be inherently static, and a **Dynamic_Entity** subclass would add operations for changing the position and orientation attributes.

Unfortunately, altering the **Entity** hierarchy in this way, plus incorporating the proposals of subsection 6.2.1.2, would be a difficult design problem. Trying to maintain the existing distinction between orientable and non-orientable entities would complicate matters further. Multiple inheritance might prove necessary.

6.2.1.5 Multiple Environments

The ObjectSim Framework currently limits each scene to a single environment. While this may seem natural at first, it is actually unnecessarily confining. What if the sky is colored

differently in different parts of the scene? Is it the same time of day in all places? Are the weather conditions uniform throughout the scene?

Since it is often difficult, if not impossible, to define the environment at every location in a scene, the best way to overcome these problems would be to allow the environment to vary according to the position and orientation of each view. That is, each view should query the scene for the local environmental conditions. As views move, they can continuously receive updates on the sky color, time of day, fog intensity, etc. **at the locations of interest.**

This adjustment could be achieved by altering the **Scene** class to keep a list of **Environment** instances, and adding a routine to return a pointer to the appropriate instance, given a position and orientation (and/or any other information). This operation would be abstract, and would represent a new tailoring point in the Framework.

6.2.2 Strategic Recommendations

6.2.2.1 Address Distributed Simulation

Adapting ObjectSim to accomodate distributed simulations in a sensible, efficient manner will be a major undertaking. Fortunately, this thesis effort shed some light on the complexities which are involved. Namely, the overall software architecture will involve a multi-process, concurrent design. Determining the high-level architecture for distributed simulations is critical at this point, and should probably be the number one priority of future work.

6.2.2.2 Convert to OpenGL

As discussed in Chapter 2, OpenGL is a standard graphics library designed by Silicon Graphics, and intended to be implemented on a wide variety of platforms. The AFIT Lab is still reliant on IRIS GL, which is the platform-dependent ancestor of OpenGL. Although OpenGL offers only a subset of GL's capabilities, it is in the Lab's best interests to convert to OpenGL and compensate for any shortcomings. The result would be application portability at the lowest possible level of abstraction.

6.2.2.3 Upgrade Low-Level Services

ObjectSim's Low-Level Services is, at present, a layer of aliases to GL and Performer constructs. This layer could be substantially improved by adding subprogram parameter defaults, ensuring identifiers are as meaningful as possible, and enhancing the in-source documentation.

Furthermore, as pointed out in Chapter 5, the LLS does not fully cover the GL and Performer libraries. Should this be remedied? Should some capabilities of the underlying libraries be intentionally left out, or collected in "optional" portions of the layer? How would the conversion to OpenGL impact LLS? Could the layer be improved by reorganizing the components? These questions must be answered before Low-Level Services reaches its full potential as a software layer which can be realistically moved to new platforms, allowing for truly platform-independent visual simulations.

6.3 Final Remarks

Like its predecessors, ObjectSim 3.0 offers developers a reusable base design for visual simulation applications. The latest version of the architecture offers the following new features:

- the capability to develop applications which take advantage of GL and Performer without actually calling those libraries directly
- a **Scene** class which captures the abstraction of the virtual world
- the capability to divide the screen into multiple viewports and dynamically attach and detach views to and from those viewports
- the capability to have entities automatically follow other entities, and to have views automatically track entities as the entities move
- a partially-specified hierarchy of simulation entities, with the capability to easily extend this hierarchy with new classes

Future efforts will continue to improve the architecture to better support the AFIT Lab. Hopefully the experiences recorded during this effort, as well as the recommendations offered in this chapter, will be beneficial to successor architects.

Appendix A. ObjectSim Application Performer References

This appendix contains the results of searching through four ObjectSim applications for references to SGI's Performer library. Any word beginning with the letters "pf" (upper, lower, or mixed case) was flagged. References within comment boundaries were ignored. The results are broken down by application. Each listing shows every Performer identifier which was referenced within that application, the number of times each identifier was flagged, and the total number of all non-comment Performer references.

A.1. The Red Flag Debriefing Tool

```
---------------------------------
Global Results for Debriefing Tool
---------------------------------
Pfmr_Renderer                   6
PFSET_VEC                       175
pfMatrix                        42
pfMakeIdentMat                  16
pfMultMat                       54
pfSqrt                          3
pfVec                           237
PFCOPY_VEC                      14
pfMalloc                        24
pfGetSharedArena                24
pfGetSemaArena                  3
pfMakeRotMat                    26
pfXformVec                      5
PFADD_VEC                       34
pfTransposeMat                  6
PFSUB_VEC                       4
pfXformPt                       14
pfNormalizeVec                  4
pfMakeScaleMat                  16
pfMakeTransMat                  19
PF_X                            177
PF_Y                            174
PF_Z                            172
pfMakeEulerMat                  16
pfNewSCS                        18
pfNewDCS                        20
pfAddChild                      58
pfGetTime                       5
PFMAKE_IDENT_MAT                2
PF_R                            14
PF_P                            23
pfDCSMat                        3
PF_H                            25
pfSetVec                        28
pfSCS                           17
pfTraverser                     3
pfChannel                       8
pfESkyColor                     3
PFES_GRND_NEAR                  1
PFES_GRND_FAR                   1
PFES_CLEAR                      1
pfESkyMode                      1
```

```
PFES_BUFFER_CLEAR               1
PFES_SKY_CLEAR                  1
pfCoord                         15
pfAddVec                        5
pfMakeCoordMat                  1
pfPreMultMat                    1
pfDotVec                        2
pfDistancePt                    4
pfDCSScale                      5
pfSinCos                        6
pfBillboard                     1
PF_DEG                          4
pfScaleVec                      2
pfNodeTravFuncs                 1
pfPushState                     1
pfBasicState                    1
pfPopState                      1
pfLengthVec                     2
pfChanViewport                  12
pfChanFOV                       8
pfDrawChanStats                 1
pfInit                          1
pfMultiprocess                  1
PFMP_APPCULLDRAW                1
pfInitClock                     2
PFNFY_INFO                      2
pfNotifyLevel                   1
pfConfig                        1
pfNodeTravMask                  4
PFTRAV_DRAW                     4
PFTRAV_SELF                     4
PF_AND                          4
pfChanNearFar                   4
pfFrameRate                     1
PFLENGTH_VEC                    6
pfDCSTrans                      1
pfSeqMode                       1
PFSEQ_START                     1
pFont                           4
pfGroup                         3
pfNewGroup                      3
pfRemoveChild                   4
PF_RAD                          6
pfDCSCoord                      2
pfz_rot                         2
pfy_rot                         2
pfx_rot                         2
pfArcSin                        4
pfCopyMat                       8
pfmr_hpr                        2
pfmr_to_euler_hpr               2
pfSeg                           4
pfMakePtsSeg                    1
pFOV                            6
pfExit                          1
pfArcTan                        1
pfArcCos                        2
pFile                           17

Performer references grand total:  1686
```

A.2. The Space Modeler

Global Results for Space Modeler	
PF_X	282
PF_Y	281
PF_Z	266
pfSeg	48
pfVec	196
PFSET_VEC	165
pfLengthVec	3
pfMatrix	33
pfCoord	36
pfMakeCoordMat	9
pfPreScaleMat	2
pfMultMat	4
pfMakeIdentMat	6
pfInvertMat	3
pfXformPt	17
pfMalloc	18
pfGetSharedArena	18
pfPushState	8
pfBasicState	8
PF_ON	5
pfPopState	10
pfTraverser	21
pfNodeTravFuncs	7
PFTRAV_DRAW	11
pfNodeTravData	5
PFTRAV_CONT	9
pfNewDCS	10
pfAddChild	41
pfDCSCoord	13
PFCOPY_VEC	69
pfMakePtsSeg	5
pfMakeScaleMat	6
pfNewSCS	5
pfGroup	6
pfGeoState	1
pfSCS	7
pfLight	1
PF_H	59
PF_P	44
PF_R	17
pfCopyVec	4
pfInit	1
pfMultiprocess	1
PFMP_DEFAULT	1
pfSetVec	1
pfArcTan	4
pfPipe	3
pfGetPipe	3
pfGetPipeSize	3
pfGetPipeOrigin	1
pfCopyMat	1
pfSwitch	5
pfNewSwitch	2
pfNewGroup	3

```
pfSwitchVal                 5
pfSegIsectTri               2
pfNode                      3
pfGetTime                   5
pfDisable                   1
PFEN_LIGHTING               1
pfChannel                   6
pfseg                       5
PFNEGATE_VEC                2
PFADD_VEC                   12
pfNormalizeVec              13
pfGetOrthoMatCoord          3
pfDrawChanStats             2
PFSUB_VEC                   16
pfAdd                       1
PFDISTANCE_PT               10
pfRemoveChild               5
pfSphere                    2
Pfmr_Renderer               1
pfuMakeTexList              1
pfChanFOV                   2
pfChanNearFar               1
pfPreMultMat                1
pfDCSMat                    1
pfLPointPos                 2
pfuDownloadTexList          2
PFUTEX_SHOW                 1
pfNode.                     1
pfNewLPoint                 1
pfLPointSize                1
pfLPointShape               1
PFLP_OMNIDIRECTIONAL        1
pfLPointColor               1
pfDCSRot                    1
pfDCS                       2
PFDOT_VEC                   2
PFLENGTH_VEC                10
PFSCALE_VEC                 10
pfList                      1
pfLightPoint                1
pfuDrawMessageRGB           10
PFU_MSG_PIPE                10
PFU_CENTER_JUSTIFIED        10
PFU_FONT_BIG                2
PFU_FONT_MED                8
pfNewLight                  1
pfLightAmbient              1
pfLightPos                  1
pfLightOn                   1
PFCOPY_MAT                  1
PFTRAV_CULL                 1
pfPreTransMat               19

Performer references grand total:  1992
```

A.3. The Synthetic Battle Bridge

Global Results for Battle Bridge	
pfSeg	79
pfVec	375
PFSET_VEC	448
PF_X	214
PF_Y	214
PF_Z	197
pfCoord	59
PF_H	61
PF_P	34
PF_R	14
pfNodeTravFuncs	5
PFTRAV_DRAW	10
pfNodeTravData	5
pfTraverser	15
pfPushState	7
pfBasicState	7
pfPopState	7
PFTRAV_CONT	5
pfMatrix	45
pfMakeCoordMat	8
pfPreScaleMat	10
pfMultMat	5
pfMakeIdentMat	13
pfInvertMat	4
pfXformPt	21
pfMalloc	34
pfGetSharedArena	37
PFCOPY_VEC	54
pfChannel	11
pfCopyVec	4
Pfmr_Renderer	8
pfInit	1
pfCopyMat	1
pfSwitch	7
pfNewSwitch	6
pfMakeScaleMat	1
pfNewSCS	1
pfNewGroup	11
pfAddChild	43
pfSCS	4
pfSwitchVal	29
pfSegIsectTri	2
pfGroup	8
pfNode	7
pfPipe	1
pfGetPipe	1
pfGetPipeSize	1
pfseg	5
pfMakePtsSeg	1
pfPreTransMat	21
pfGetOrthoMatCoord	3
pfDCSCoord	5
pfDrawChanStats	2
pfRemoveChild	8

PFSWITCH_ON	17
PFSWITCH_OFF	17
pfGetTime	3
pfNewDCS	12
pfNewLOD	1
pfLODRange	3
pfInsertChild	6
PF_RAD	3
PFLENGTH_VEC	3
pfNormalizeVec	2
PFSCALE_VEC	2
pfGetSwitchVal	12
pf	7
pfDCSMat	7
PFSUB_VEC	4
pfLOD	1
pfChanLODAttr	1
PFLOD_FADE	1
pfNewFog	2
pfFogType	2
PFFOG_PIX_EXP	2
pfChanViewport	2
pfFog	6
pfDCSTrans	2
pfSeqMode	4
PFSEQ_START	4
PFDISTANCE_PT	2
PFCOPY_MAT	1
pfEarthSky	5
pfLight	5
pfNewLight	1
pfLightPos	2
pfLightAmbient	2
pfLightColor	2
pfDCSScale	6
pfNode.	1
pfNewLPoint	1
pfLPointShape	1
PFLP_OMNIDIRECTIONAL	1
pfLPointColor	3
pfLPointSize	1
pfLPointPos	1
pfOverride	3
PFSTATE_FOG	3
PFSTATE_ENFOG	3
PF_OFF	3
pfFogRange	1
pfFogColor	1
pfEnable	1
PFEN_FOG	2
pfApplyFog	1
PF_ON	1
pfDisable	1
pfESkyAttr	3
PFES_GRND_HT	2
pfESkyMode	2
PFES_BUFFER_CLEAR	2
PFES_FAST	1
pfESkyColor	6
PFES_CLEAR	1

```
pfChanESky                    2
PFES_SKY_GRND                 1
PFES_SKY_TOP                  1
PFES_SKY_BOT                  1
PFES_HORIZ                    1
PFES_GRND_NEAR                1
PFES_GRND_FAR                 1
PFES_HORIZ_ANGLE              1
pfLightOff                    1
pfGetCurLights                1
pfDCS                         4
pfBillboard                   1
pfLightPoint                  1

Performer references grand total:  2394
```

A.4. The Virtual Cockpit

```
---------------------------------
Global Results for Virtual Cockpit
---------------------------------
pfTraverser                   12
pfMalloc                      22
pfGetSharedArena              22
pfCoord                       21
pfMatrix                      72
pfMakeEulerMat                60
pfNewSCS                      65
pfNewDCS                      64
pfNewGroup                    2
pfAddChild                    209
PFSET_VEC                     405
pfNodeTravFuncs               4
pfNodeTravData                4
PFTRAV_DRAW                   14
pfSeg                         56
pfVec                         629
pfChannel                     7
pfPushState                   8
pfBasicState                  8
pfPopState                    8
PFTRAV_CONT                   4
pfDrawChanStats               1
PF_X                          184
PF_Y                          183
PF_Z                          160
pfScaleVec                    10
PFADD_VEC                     92
PF_H                          35
PF_P                          23
PF_R                          18
PFCOPY_VEC                    71
pfDCSCoord                    10
pfseg                         5
pfMakePtsSeg                  11
pfGroup                       3
pfSCS                         35
Pfmr_Renderer                 4
pfMakeTransMat                72
```

pfRemoveChild	16
pfGetTime	21
pfMakeRotMat	49
pfMultMat	167
pfXformVec	22
pfuIsect	10
pfuSegsIsectNode	9
PFTRAV_IS_PRIM	9
PFSUB_VEC	15
pfSetVec	42
pfLengthVec	8
pfArcTan	11
pfMakeCoordMat	6
pfPreScaleMat	5
pfMakeIdentMat	18
pfInvertMat	6
pfXformPt	64
pfmr	1
pfSegsIsectNode	7
pfPlane	2
PFNEGATE_VEC	1
pfMakeNormPtPlane	2
pfPtInHalfSpace	2
PFTRAV_IS_GEODE	1
PFTRAV_IS_CULL_BACK	1
pfCopyVec	22
pfGetSemaArena	7
pfSqrt	6
pfTransposeMat	2
pfNormalizeVec	3
pfSinCos	3
pfArcCos	1
pfSubVec	9
pfDotVec	2
pfMakePolarSeg	1
pfDistancePt	8
pfPtInFrust	2
pfFrustNearFar	2
pfMakeSimpleFrust	4
pfOrthoXformFrust	2
pfNewFrust	2
pfFrustum	2
pfMakeScaleMat	67
PFMAKE_IDENT_MAT	2
pfDCSMat	3
pfDCSRot	79
pfDCSTrans	12
pfInit	1
pfMultiprocess	1
PFMP_DEFAULT	1
pfEarthSky	2
PFES_SKY_TOP	2
PFES_SKY_BOT	2
PFES_HORIZ	2
pfNewESky	1
pfESkyColor	9
PFES_GRND_NEAR	2
PFES_GRND_FAR	2
PFES_CLEAR	2
pfESkyMode	1

```
PFES_BUFFER_CLEAR               1
PFES_SKY_CLEAR                  1
pfESkyAttr                      1
PFES_HORIZ_ANGLE                1
pfChanESky                      1
pfChanTravMode                  7
PFDRAW_OFF                      5
pfChanFOV                       2
PFDRAW_ON                       2
pfGetChanESky                   1
pfChanViewport                  2
PFSQR_DISTANCE_PT               1
pfCopyMat                       1
pfuMakeTexList                  1
pfNode                          2
pfuDownloadTexList              1
PFUTEX_SHOW                     1
pfList                          1
pfvec                           1
pfmr_renderer                   1
pfSwitch                        2
pfNewSwitch                     1
pfSwitchVal                     1
pfSegIsectTri                   2
pfInitClock                     1
PFSCALE_VEC                     1
PFCOPY_MAT                      1

Performer references grand total:  3420
```

In total, the Ada 95 implementation of ObjectSim 3.0 currently exceeds 6,000 lines of source (including comments and blank lines). For the sake of space, this appendix contains only the package specifications for the Ada 95 Application Framework.

B.1. The all-encompassing Framework package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework
--   Author: Capt Shawn Hannan
--
-- Comments: This package collects all aspects of the framework in one place.
--
--  History: 23 Sep 95 - created
--
------------------------------------------------------------------------
with ObjectSim.Low_Level_Services; use ObjectSim.Low_Level_Services;
package ObjectSim.Framework is

end ObjectSim.Framework;
```

B.2. The Coordinate System package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Coordinate_System
--   Author: Capt Shawn Hannan
--
-- Comments: This package collects all constants, types and operations
--           related to the world coordinate system supported by the
--           ObjectSim Framework.
--
--           Distance and Angle should really be private types, but are
--           currently implemented as subtypes of other visible types.
--
--  History: 22 Sep 95 - created from ObjectSim.API.Coordinate_System
--
------------------------------------------------------------------------
with ObjectSim.Low_Level_Services.Math_Utilities;
package ObjectSim.Framework.Coordinate_System is

   X : constant := Math_Utilities.X;
   Y : constant := Math_Utilities.Y;
   Z : constant := Math_Utilities.Z;

   Heading : constant := Math_Utilities.Heading;
   Pitch   : constant := Math_Utilities.Pitch;
   Roll    : constant := Math_Utilities.Roll;

   subtype Distance is Math_Utilities.Distance_Type;
   subtype Angle is Math_Utilities.Angle_Type;

   subtype Position_Vector is Math_Utilities.Vector_3d_Type;
   subtype Orientation_Vector is Math_Utilities.Vector_3d_Type;

   -- can be used for position vectors or orientation vectors
   function "+" (Left, Right : Math_Utilities.Vector_3d_Type)
      return Math_Utilities.Vector_3d_Type;

end ObjectSim.Framework.Coordinate_System;
```

B.3. The abstract Environment package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Environment
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's Environment class
--
--  History: 22 Sep 95 - created from ObjectSim.API.Environment
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Low_Level_Services.Vis_Sim;
package ObjectSim.Framework.Environment is

   type Object is abstract new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   -- For setting the percentage of Red, Green and Blue in a color
   type Intensity_Percentage is digits 7 range 0.0 .. 1.0;

   --
   -- The abstract Update operation
   --
   procedure Update (Instance : in out Object) is abstract;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   --
   -- Background color operations
   --
   procedure Set_Background_Color (Instance : in out Object;
                                   Red,
                                   Green,
                                   Blue,
                                   Alpha    : Intensity_Percentage);

   function Background_Red_Of (Instance : Object) return Intensity_Percentage;

   function Background_Green_Of (Instance : Object) return Intensity_Percentage;

   function Background_Blue_Of (Instance : Object) return Intensity_Percentage;

   function Background_Alpha_Of (Instance : Object) return Intensity_Percentage;

   --
   -- Low_Level_Services operations
   --
   function Get_Earthsky (Instance : Object) return Vis_Sim.Earthsky_Type;

private

   type Object is abstract new Ada.Finalization.Limited_Controlled with
      record
         Earthsky : Vis_Sim.Earthsky_Type;
         Red,
         Green,
         Blue,
         Alpha    : Intensity_Percentage := 0.0;
      end record;

end ObjectSim.Framework.Environment;
```

B.4. The abstract Entity package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Entity
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's Entity class
--
--           An entity is anything which can be placed in a scene.
--
--  History: 22 Sep 95 - created from ObjectSim.API.Entity
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Low_Level_Services.Vis_Sim;
package ObjectSim.Framework.Entity is

   type Object is abstract new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   --
   -- Low_Level_Services operations
   --
   function Get_Node (Instance : Object) return Vis_Sim.Dynamic_Coordinate_System_Type;

private

   -- I am making an important simplifying assumption here.  I am assuming
   -- every entity can be positioned within a scene.  Furthermore, most
   -- entities will have an associated orientation.  I am therefore using
   -- the Vis_Sim.Dynamic_Coordinate_System_Type for all entities.  This
   -- will adversely affect performance, but will ease implementation.

   type Object is abstract new Ada.Finalization.Limited_Controlled with
      record
         Node     : Vis_Sim.Dynamic_Coordinate_System_Type;
      end record;

end ObjectSim.Framework.Entity;
```

B.5. The abstract Orientable Entity package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Entity.Orientable
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's Orientable Entity class
--
--           An orientable entity is any entity which has a heading, pitch
--           and roll.
--
--  History: 22 Sep 95 - created from ObjectSim.API.Entity.Orientable
--
------------------------------------------------------------------------
with ObjectSim.Framework.Coordinate_System; use ObjectSim.Framework.Coordinate_System;
package ObjectSim.Framework.Entity.Orientable is

   type Object is abstract new Entity.Object with private;

   type Reference is access all Object'Class;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);
```

```
   procedure Finalize (Instance : in out Object);

   --
   -- Orientation operations
   --
   procedure Set_Orientation (Instance               : in out Object;
                              New_Orientation_Vector : in     Orientation_Vector);

   procedure Set_Heading (Instance    : in out Object;
                          New_Heading : in     Angle);

   procedure Set_Pitch (Instance  : in out Object;
                        New_Pitch : in     Angle);

   procedure Set_Roll (Instance : in out Object;
                       New_Roll : in     Angle);

   function Orientation_Of (Instance : Object) return Orientation_Vector;

   function Heading_Of (Instance : Object) return Angle;

   function Pitch_Of (Instance : Object) return Angle;

   function Roll_Of (Instance : Object) return Angle;

private

   type Object is abstract new Entity.Object with
      record
         Heading,
         Pitch,
         Roll    : Angle := 0.0;
      end record;

end ObjectSim.Framework.Entity.Orientable;
```

B.6. The abstract 3D Entity package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Entity.Orientable.ThreeD
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's ThreeD model class
--
--  History: 22 Sep 95 - created from ObjectSim.API.Entity.Orientable.ThreeD
--
------------------------------------------------------------------------
package ObjectSim.Framework.Entity.Orientable.ThreeD is

   type Object is abstract new Orientable.Object with private;

   type Reference is access all Object'Class;

   procedure Update (Instance : in out Object) is abstract;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   --
   -- File operations
   --
   procedure Load (Instance : in out Object;
                   Filename : in     String);

private

   type Object is abstract new Orientable.Object with null record;

end ObjectSim.Framework.Entity.Orientable.ThreeD;
```

B.7. *The abstract Scene package*

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Scene
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's Scene class
--
--  History: 22 Sep 95 - created from ObjectSim.API.Scene
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Low_Level_Services.Vis_Sim;
with ObjectSim.Framework.Coordinate_System; use ObjectSim.Framework.Coordinate_System;
with ObjectSim.Framework.Entity.Orientable;
with ObjectSim.Framework.Environment;
package ObjectSim.Framework.Scene is

   type Object is abstract new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   --
   -- The abstract Update operation
   --
   procedure Update (Instance : in out Object) is abstract;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   --
   -- Boundary operations
   --
   procedure Set_Boundaries (Instance       : in out Object;
                             Minimum_Corner : in     Position_Vector;
                             Maximum_Corner : in     Position_Vector);

   function Minimum_Corner_Of (Instance : Object) return Position_Vector;

   function Maximum_Corner_Of (Instance : Object) return Position_Vector;

   --
   -- Environment operations
   --
   procedure Set_Environment (Instance        : in out Object;
                              New_Environment : in     Environment.Reference);

   function Environment_Of (Instance : Object) return Environment.Reference;

   --
   -- Entity operations
   --
   procedure Add (Instance   : in out Object;
                  New_Entity : in     Entity.Reference);

   procedure Delete (Instance      : in out Object;
                     Target_Entity : in     Entity.Reference);

   procedure Move (Instance      : in out Object;
                   Target_Entity : in     Entity.Reference;
                   Offset        : in     Position_Vector);

   procedure Set_Position (Instance      : in out Object;
                           Target_Entity : in     Entity.Reference;
                           New_Position  : in     Position_Vector);

   procedure Move_Straight (Instance      : in out Object;
                            Target_Entity : in     Entity.Orientable.Reference;
                            How_Far       : in     Distance);
```

```
   procedure Look_At (Instance      : in out Object;
                      Target_Entity : in     Entity.Orientable.Reference;
                      Position      : in     Position_Vector);

   function Position_Of (Instance      : Object;
                         Target_Entity : Entity.Reference) return Position_Vector;

   function Scene_Position_Of (Instance      : Object;
                               Target_Entity : Entity.Reference) return Position_Vector;

   function Number_Of_Entities_In (Instance : Object) return Natural;

   function Get_Entity (Instance : Object;
                        Number   : Positive) return Entity.Reference;

   procedure Follow (Instance   : in out Object;
                     Follower   : in     Entity.Reference;
                     Leader     : in     Entity.Reference);

   procedure Stop_Following (Instance      : in out Object;
                             Follower      : in     Entity.Reference);

   function Is_Following (Instance      : Object;
                          Target_Entity : Entity.Reference) return Boolean;

   function Number_Of_Followers (Instance      : Object;
                                 Target_Entity : Entity.Reference)
      return Natural;

   function Leader_Of (Instance      : Object;
                       Target_Entity : Entity.Reference)
      return Entity.Reference;

   function Get_Follower (Instance      : Object;
                          Target_Entity : Entity.Reference;
                          Number        : Positive)
      return Entity.Reference;

   --
   -- Low_Level_Services operations
   --
   function Get_Scene_Root (Instance : Object) return Vis_Sim.Scene_Type;

   --
   -- Exceptions
   --

   -- raised when an entity operation is called for an entity
   -- which has not been added to the scene
   Entity_Not_In_Scene_Error : exception;

   -- raised when Stop_Following or Leader_Of is called, but
   -- entity is not following anything
   Not_Following_Error : exception;

   -- raised by Get_Entity or Get_Follower when Number is
   -- too high
   Index_Error : exception;

   -- raised by Move or Set_Position if position adjustment
   -- would move entity outside boundaries
   Position_Error : exception;

   -- raised by Environment_Of if there is the environment of
   -- the scene has not been set
   No_Environment_Error : exception;

private

   -- a rather arbitrary "maximum distance" used to establish default corners
   -- for the scene
   Max_Distance : constant := 1_000_000.0;
```

```
   Default_Minimum_Corner : constant Position_Vector := (X => -Max_Distance,
                                                         Y => -Max_Distance,
                                                         Z => -Max_Distance);

   Default_Maximum_Corner : constant Position_Vector := (X => Max_Distance,
                                                         Y => Max_Distance,
                                                         Z => Max_Distance);

   -- Better to use linked lists, but I'll use arrays for now.

   Max_Entities_Per_Scene   : constant := 50;
   Max_Followers_Per_Entity : constant := 20;

   type Follower_List is array (1 .. Max_Followers_Per_Entity)
      of Entity.Reference;

   type Entity_Record is
      record
         Number_Of_Followers : Natural          := 0;
         Entity_Ptr          : Entity.Reference;
         Position            : Position_Vector  := (0.0,0.0,0.0);
         Followers           : Follower_List;
         Following           : Entity.Reference := null; -- not following anyone
                                                         -- when null
      end record;

   type Entity_List is array (1 .. Max_Entities_Per_Scene) of Entity_Record;

   type Object is abstract new Ada.Finalization.Limited_Controlled with
      record
         Number_Of_Entities  : Natural               := 0;
         Entities            : Entity_List;
         Scene               : Vis_Sim.Scene_Type;
         Minimum_Corner      : Position_Vector       := Default_Minimum_Corner;
         Maximum_Corner      : Position_Vector       := Default_Maximum_Corner;
         Environment_Ptr     : Environment.Reference;
      end record;

end ObjectSim.Framework.Scene;
```

B.8. The abstract View package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.View
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's View class
--
--  History: 22 Sep 95 - created from ObjectSim.API.View
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Low_Level_Services.Vis_Sim;
with ObjectSim.Framework.Scene;
with ObjectSim.Framework.Entity;
with ObjectSim.Framework.Coordinate_System; use ObjectSim.Framework.Coordinate_System;
package ObjectSim.Framework.View is

   type Object is abstract new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   --
   -- The abstract Update operation
   --
   procedure Update (Instance : in out Object) is abstract;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);
```

```
--
-- Scene operations
--
procedure Set_Scene (Instance     : in out Object;
                     Target_Scene : in     Scene.Reference);

function Scene_Of (Instance : Object) return Scene.Reference;

--
-- Field of view operations
--
procedure Set_Horizontal_FOV (Instance  : in out Object;
                              New_Angle : in     Angle  := -1.0);

procedure Set_Vertical_FOV (Instance  : in out Object;
                            New_Angle : in     Angle  := -1.0);

function Horizontal_FOV_Of (Instance : Object) return Angle;

function Vertical_FOV_Of (Instance : Object) return Angle;

--
-- Clipping plane operations
--
procedure Set_Near_Clipping_Plane_Distance (Instance     : in out Object;
                                            New_Distance : in     Distance);

procedure Set_Far_Clipping_Plane_Distance (Instance     : in out Object;
                                           New_Distance : in     Distance);

function Near_Clipping_Plane_Distance_Of (Instance : Object) return Distance;

function Far_Clipping_Plane_Distance_Of (Instance : Object) return Distance;

--
-- Position/Orientation operations
--
procedure Set_Position (Instance     : in out Object;
                        New_Position : in     Position_Vector);

procedure Move (Instance : in out Object;
                Offset   : in     Position_Vector);

procedure Set_Orientation (Instance        : in out Object;
                           New_Orientation : in     Orientation_Vector);

function Orientation_Of (Instance : Object) return Orientation_Vector;

function Position_Of (Instance : Object) return Position_Vector;

function Scene_Position_Of (Instance : Object) return Position_Vector;

procedure Follow (Instance      : in out Object;
                  Target_Entity : in     Entity.Reference);

procedure Stop_Following (Instance : in out Object);

function Is_Following (Instance : Object) return Boolean;

function Leader_Of (Instance : Object) return Entity.Reference;

procedure Look_At (Instance : in out Object;
                   Position : in     Position_Vector);

procedure Track (Instance      : in out Object;
                 Target_Entity : in     Entity.Reference);

procedure Stop_Tracking (Instance : in out Object);

function Is_Tracking (Instance : Object) return Boolean;

function Trackee_Of (Instance : Object) return Entity.Reference;
```

```
   --
   -- Copy operations
   --
   procedure Mimic (Original : in     Object;
                    Copy     : in out Object);

   --
   -- Low_Level_Services operations
   --
   function Get_Channel (Instance : Object) return Vis_Sim.Channel_Type;

   --
   -- Exceptions
   --

   -- raised by Move or Set_Position if position adjustment would
   -- move view outside scene boundaries
   Position_Error : exception;

   -- raised by Stop_Following and Leader_Of when the view is not
   -- following an entity
   Not_Following_Error : exception;

   -- raised by Stop_Tracking and Trackee_Of when the view is not
   -- tracking an entity
   Not_Tracking_Error : exception;

private

   type Object is abstract new Ada.Finalization.Limited_Controlled with
      record
         Channel                       : Vis_Sim.Channel_Type;
         Scene_Ptr                     : Scene.Reference;
         Horizontal_FOV,
         Vertical_FOV                  : Angle                := 90.0;
         Near_Clipping_Plane_Distance  : Distance             := 1.0;
         Far_Clipping_Plane_Distance   : Distance             := 100_000.0;
         Position                      : Position_Vector      := (0.0,0.0,0.0);
         Orientation                   : Orientation_Vector   := (0.0,0.0,0.0);
         Following                     : Entity.Reference     := null;
         Trackee                       : Entity.Reference     := null;
      end record;

end ObjectSim.Framework.View;
```

B.9. The private View Movement package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.View.Movement
--   Author: Capt Shawn Hannan
--
-- Comments: a private package for use by other Framework class implementations
--
--           Specifically, this package was initially created as a place to
--           put the Catch_Up operation.  This operation will be called each
--           frame by the Renderer, but there is no need for this operation
--           to be publicly available.
--
--  History: 22 Sep 95 - created from ObjectSim.API.View.Movement
--
------------------------------------------------------------------------
private package ObjectSim.Framework.View.Movement is

   procedure Catch_Up (Instance : in out Object);

end ObjectSim.Framework.View.Movement;
```

B.10. The Renderer Package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Renderer
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim Framework's Renderer object
--
--           The Renderer manipulates a single implied window.  Within
--           that window, any number of viewports may be created.  Each
--           viewport is capable of displaying the output for a single
--           ObjectSim view.
--
--  History: 22 Sep 95 - created from ObjectSim.API.Renderer
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Framework.View;
with ObjectSim.Low_Level_Services.Vis_Sim;
package ObjectSim.Framework.Renderer is

   Max_Frame_Rate : constant := 60.0;
   Max_Viewports  : constant := 4;

   type Viewport_Count is range 0 .. Max_Viewports;
   subtype Viewport_Index is Viewport_Count range 1 .. Viewport_Count'Last;

   type Frame_Rate is digits 7 range 0.0 .. Max_Frame_Rate;

   -- Used to set up viewport sizes
   type Normalized_Distance is digits 7 range 0.0 .. 1.0;

   --
   -- Initialize and Finalize operations
   --
   procedure Initialize;

   procedure Finalize;

   --
   -- Frame rate operations
   --
   procedure Set_Frame_Rate (New_Frame_Rate : in     Frame_Rate);

   function Current_Frame_Rate return Frame_Rate;

   --
   -- Draw operations
   --
   procedure Synchronize;

   procedure Draw_Frames;


   --
   -- Viewport operations
   --
   function New_Viewport return Viewport_Index;

   function Number_Of_Viewports return Viewport_Count;

   procedure Set_Corners (Viewport_Number : in     Viewport_Index;
                          Left            : in     Normalized_Distance := 0.0;
                          Right           : in     Normalized_Distance := 1.0;
                          Bottom          : in     Normalized_Distance := 0.0;
                          Top             : in     Normalized_Distance := 1.0);

   procedure Set_View (Viewport_Number : in     Viewport_Index;
                       Target_View     : in     View.Reference);

   function Left_Of (Viewport_Number : Viewport_Index)
      return Normalized_Distance;
```

```
      function Right_Of (Viewport_Number : Viewport_Index)
         return Normalized_Distance;

      function Bottom_Of (Viewport_Number : Viewport_Index)
         return Normalized_Distance;

      function Top_Of (Viewport_Number : Viewport_Index)
         return Normalized_Distance;

      function Associated_View_Of (Viewport_Number : Viewport_Index)
         return View.Reference;

      function Associated_Viewport_Of (Target_View : View.Reference)
         return Viewport_Index;

      --
      -- Low_Level_Services operations
      --
      function Get_Pipe return Vis_Sim.Pipe_Type;

      --
      -- Exceptions
      --

      -- raised by New_Viewport when max viewports already allocated
      Too_Many_Viewports_Error : exception;

      -- raised by Draw_Frames when no views have been set
      No_Views_Error : exception;

      -- raised by Associated_Viewport_Of when there is no associated viewport
      No_Associated_Viewport_Error : exception;

      -- raised by Associated_View_Of when there is no associated view
      No_Associated_View_Error : exception;

   end ObjectSim.Framework.Renderer;
```

B.11. The Scene_Manager package

```
------------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Scene_Manager
--   Author: Capt Shawn Hannan
--
-- Comments: Manages instances of the Framework's Scene class
--
--           This is a concrete class.  The default behavior is simple.
--           Developers "register" their scenes with the Scene_Manager by
--           calling Add.  Subsequent calls to Update will cause the
--           Scene_Manager to call the Update operation for each scene.
--
--  History: 22 Sep 95 - created
--
------------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Framework.Scene;
package ObjectSim.Framework.Scene_Manager is

   -- Arbitrarily set to 3 for now
   Max_Scenes : constant := 3;

   type Object is new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   procedure Add (Instance  : in out Object;
                  New_Scene : in     Scene.Reference);

   procedure Update (Instance : in out Object);

   -- raised by Add when adding a new scene would exceed the maximum
   Too_Many_Scenes_Error : exception;
```

```
private

   type Scene_Count is range 0 .. Max_Scenes;
   subtype Scene_Index is Scene_Count range 1 .. Max_Scenes;

   type Scene_List is array (Scene_Index) of Scene.Reference;

   type Object is new Ada.Finalization.Limited_Controlled with
      record
         Number_Of_Scenes : Scene_Count := 0;
         Scenes           : Scene_List;
      end record;

end ObjectSim.Framework.Scene_Manager;
```

B.11. The View_Manager package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.View_Manager
--   Author: Capt Shawn Hannan
--
-- Comments: Manages instances of the Framework's View class
--
--           This is a concrete class.  The default behavior is simple.
--           Developers "register" their views with the View_Manager by
--           calling Add.  Subsequent calls to Update will cause the
--           View_Manager to call the Update operation for each view.
--
--  History: 22 Sep 95 - created
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Framework.View;
package ObjectSim.Framework.View_Manager is

   -- Arbitrarily set to 8 for now
   Max_Views : constant := 8;

   type Object is new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   procedure Add (Instance : in out Object;
                  New_View : in     View.Reference);

   procedure Update (Instance : in out Object);

   -- raised by Add when adding a new view would exceed the maximum
   Too_Many_Views_Error : exception;

private

   type View_Count is range 0 .. Max_Views;
   subtype View_Index is View_Count range 1 .. Max_Views;

   type View_List is array (View_Index) of View.Reference;

   type Object is new Ada.Finalization.Limited_Controlled with
      record
         Number_Of_Views : View_Count := 0;
         Views           : View_List;
      end record;

end ObjectSim.Framework.View_Manager;
```

B.13. The abstract Simulation package

```
------------------------------------------------------------------------
--     Unit: ObjectSim.Framework.Simulation
--   Author: Capt Shawn Hannan
--
-- Comments: The ObjectSim framework's Simulation class
--
--           This abstract class serves as the template for all ObjectSim
--           visual simulations which are based on the Framework.
--
--  History: 22 Sep 95 - created
--
------------------------------------------------------------------------
with Ada.Finalization;
with ObjectSim.Framework.Scene_Manager;
with ObjectSim.Framework.View_Manager;
package ObjectSim.Framework.Simulation is

   type Object is abstract new Ada.Finalization.Limited_Controlled with private;

   type Reference is access all Object'Class;

   -- Default behavior for Initialize and Finalize is provided.  This behavior
   -- is mandatory.  Subclasses which override these operations must call the
   -- superclass operations as part of their implementations.
   procedure Initialize (Instance : in out Object);

   procedure Finalize (Instance : in out Object);

   procedure Visualize (Instance : in out Object);

   --
   -- Manager retrieval operations
   --
   function Scene_Manager_Of (Instance : Object) return Scene_Manager.Reference;

   function View_Manager_Of (Instance : Object) return View_Manager.Reference;

private

   -- Subclass designers may add any number of scenes, views and
   -- entities, as well as an optional environment.

   type Object is abstract new Ada.Finalization.Limited_Controlled with
      record
         Scene_Manager1 : Scene_Manager.Reference;
         View_Manager1  : View_Manager.Reference;
      end record;

end ObjectSim.Framework.Simulation;
```

Appendix C. Low-Level-Services Cross-Reference Tables

The following tables provide cross-references from ObjectSim 3.0 Low-Level-Services routines to their SGI counterpart routines. Known omissions are italicized and listed at the end of each table.

C.1. Math Utilities (SGI Counterpart: libprmath)

Low-Level-Services Routine	SGI Routine
SinCos	Pfsincos
Arctan	Pfarctan2
Arcsin	Pfarcsin
Arccos	Pfarccos
Degrees_To_Radians	PF_DEG2RAD
Radians_To_Degrees	PF_RAD2DEG
Max	PF_MAX2
Min	PF_MIN2
Set_Vector_2d	Pfsetvec2
Subtract_Vector_2d	Pfsubvec2
Length_Of_Vector_2d	Pflengthvec2
Set_Vector_3d	Pfsetvec3
Copy_Vector_3d	Pfcopyvec3
Negate_Vector_3d	Pfnegatevec3
Add_Vector_3d	Pfaddvec3
Subtract_Vector_3d	Pfsubvec3
Scale_Vector_3d	Pfscalevec3
Normalize_Vector_3d	Pfnormalizevec3
Cross_Multiply_Vector_3d	Pfcrossvec3
Transform_Point_3d	Pfxformpt3
Length_Of_Vector_3d	Pflengthvec3
Distance_Between_3d_Points	Pfdistancept3
Make_Segment	Pfmakeptsseg
Make_Polar_Segment	Pfmakepolarseg
New_Frustum	Pfnewfrust
Make_Empty_Box	Pfmakeemptybox
Make_Euler_Matrix	Pfmakeeulermat
Make_Box_Around_Points	Pfboxaroundpts
Make_Translation_Matrix	Pfmaketransmat
Make_Scaling_Matrix	Pfmakescalemat
Make_Coordinate_Matrix	Pfmakecoordmat
Get_Orthogonal_Matrix_Coordinates	Pfgetorthomatcoord
Multiply_Matrices	Pfmultmat
Post_Multiply_Matrix	Pfpostmultmat
Pre_Multiply_Matrix	Pfpremultmat
Invert_Matrix	Pfinvertmat

C.2. Rendering (SGI Counterpart: libpr)

Low-Level-Services Routine	SGI Routine
New_Geostate	Pfnewgstate
Set_Attributes	Pfgstateattr
Push_State	Pfpushstate
Basic_State	Pfbasicstate
Pop_State	Pfpopstate
Override	Pfoverride
Set_Cullface	Pfcullface
Enable	Pfenable
Disable	Pfdisable
New_Geoset	Pfnewgset
Set_Number_Of_Primitives	Pfgsetnumprims
Number_Of_Primitives_Of	Pfgetgsetnumprims
Set_Primitive_Type	Pfgsetprimtype
Primitive_Type_Of	Pfgetgsetprimtype
Set_Primitive_Lengths	Pfgsetprimlengths
Primitive_Lengths_Of	Pfgetgsetprimlengths
Set_Geostate	Pfgsetgstate
Geostate_Of	Pfgetgsetgstate
Set_Attributes	Pfgsetattr
Attribute_Binding_Of	Pfgetgsetattrbind
Get_Attribute_Lists_Of	Pfgetgsetattrlists
Set_Draw_Mode	Pfgsetdrawmode
Draw_Mode_Of	Pfgetgsetdrawmode
Draw	Pfdrawgset
Set_Line_Width	Pfgsetlinewidth
Line_Width_Of	Pfgetgsetlinewidth
Set_Point_Size	Pfgsetpntsize
Point_Size_Of	Pfgetgsetpntsize
Get_Bounding_Box	Pfgetgsetbbox
Set_Bounding_Box	Pfgsetbbox
New_Texture	Pfnewtex
Load_Texture_File	Pfloadtexfile
Free_Texture_Image	Pffreeteximage
Texture_Image_Of	Pfteximage
Set_Texture_Format	Pftexformat
Texture_Format_Of	Pfgettexformat
Set_Texture_Filter	Pftexfilter
Texture_Filter_Of	Pfgettexfilter
Set_Texture_Repeat	Pftexrepeat
Texture_Repeat_Of	Pfgettexrepeat
Apply_Texture	Pfapplytex
Idle_Texture	Pfidletex
Is_Texture_Loaded	Pfistexloaded
Current_Texture	Pfgetcurtex
New_Texture_Environment	Pfnewtenv
Apply_Texture_Environment	Pfapplytenv
Set_Texture_Environment_Mode	Pftenvmode
New_Material	Pfnewmtl
Set_Side	Pfmtlside
Side_Of	Pfgetmtlside

Set_Alpha	Pfmtlalpha
Alpha_Of	Pfgetmtlalpha
Set_Shininess	Pfmtlshininess
Shininess_Of	Pfgetmtlshininess
Set_Color	Pfmtlcolor
Set_Color_Mode	Pfmtlcolormode
Apply_Material	Pfapplymtl
New_Light	Pfnewlight
New_Light_Model	Pfnewlmodel
Set_Ambience	Pflmodelambient
Apply_Light_Model	Pfapplylmodel
New_Fog	Pfnewfog
Set_Type	Pffogtype
Set_Range	Pffogrange
Get_Range_Of	Pfgetfogrange
Set_Offsets	Pffogoffsets
Set_Ramp	Pffogramp
Set_Color	Pffogcolor
Apply_Fog	Pfapplyfog
Current_Fog	Pfgetcurfog
Push_Identity_Matrix	Pfpushidentmatrix
Pop_Matrix	Pfpopmatrix
Malloc	Pfmalloc
Free	Pffree
Set_Notify_Level	Pfnotifylevel
Current_Notify_Level	Pfgetnotifylevel
Get_Time	Pfgettime
Initialize_Clock	Pfinitclock
Set_File_Path	Pffilepath
Get_Shared_Arena	Pfgetsharedarena
not covered	*Pflightambient*
not covered	*Pflightcolor*
not covered	*Pflightpos*
not covered	*Pflighton*
not covered	*Pfnewgeoset*
not covered	*Pfdeltex*
not covered	*Pftexdeletor*
not covered	*Pfcopytex*

C.3. Import Utilities (SGI Counterpart: libpfflt/libpfsgi)

Low-Level-Services Routine	**SGI Routine**
Load_Flt	Loadflt
Load_Any	Loadfile

C.4. Vis_Sim (SGI Counterpart: libpf)

Low-Level-Services Routine	SGI Routine
Initialize	Pfinit
Cleanup	Pfexit
Multiprocess	Pfmultiprocess
Multipipe	Pfmultipipe
Configure	Pfconfig
Synchronize	Pfsync
Frame	Pfframe
Draw	Pfdraw
Cull	Pfcull
Set_Isector_Function	Pfisectfunc
Set_Synchronization_Method	Pfphase
Set_Frame_Rate	Pfframerate
Frame_Rate	Pfgetframerate
Field_Rate	Pfgetfieldrate
Initialize_Pipe	Pfinitpipe
Get_Pipe	Pfgetpipe
Initialize_Graphics	Pfinitgfx
Get_Pipe_Origin	Pfgetpipeorigin
Get_Pipe_Size	Pfgetpipesize
Get_Channel_Origin	Pfgetchanorigin
Get_Channel_Size	Pfgetchansize
New_Channel	Pfnewchan
Set_Viewport	Pfchanviewport
Get_Viewport	Pfgetchanviewport
Set_Viewpoint	Pfchanview
Set_Traversal_Mode	Pfchantravmode
Set_Clipping_Planes	Pfchannearfar
Set_Field_Of_View	Pfchanfov
Allocate_Data	Pfallocchandata
Pass_Data	Pfpasschandata
Set_Draw_Function	Pfchandrawfunc
Set_Cull_Function	Pfchancullfunc
Attach_Channel_To_Scene	Pfchanscene
Draw_Statistics	Pfdrawchanstats
Clear	Pfclearchan
Set_Stress_Filter	Pfchanstressfilter
Set_Stress	Pfchanstress
Stress_Of	Pfgetchanstress
Load_Of	Pfgetchanload
Attach_Earthsky_To_Channel	Pfchanesky
Set_Level_Of_Detail_Attributes	Pfchanlodattr
New_Earthsky	Pfnewesky
Set_Mode	Pfeskymode
Set_Attribute	Pfeskyattr
Set_Color	Pfeskycolor
Set_Name	Pfnodename
Name_Of	Pfgetnodename
Parent_Of	Pfgetparent
Number_Of_Parents_Of	Pfgetnumparents
Clone	Pfclone

Flatten	Pfflatten
Set_Traversal_Functions	Pfnodetravfuncs
Set_Traversal_Data	Pfnodetravdata
New_Group	Pfnewgroup
Find_Group	Pffindgroup
Add_Child	Pfaddchild
Replace_Child	Pfreplacechild
Insert_Child	Pfinsertchild
Remove_Child	Pfremovechild
Get_Child	Pfgetchild
Number_Of_Children_Of	Pfgetnumchildren
New_Scene	Pfnewscene
New_Dynamic_Coordinate_System	Pfnewdcs
Set_Scale	Pfdcsscale
Set_Rotation	Pfdcsrot
Set_Translation	Pfdcstrans
New_Level_Of_Detail	Pfnewlod
Find_Level_Of_Detail	Pffindlod
Set_Level_Of_Detail_Range	Pflodrange
Set_Level_Of_Detail_Center	Pflodcenter
Get_Level_Of_Detail_Center	Pfgetlodcenter
New_Light	Pfnewlsource
Set_Ambient_Light	Pflightambient
Set_Light_Color	Pflightcolor
Set_Light_Position	Pflightpos
Turn_On	Pflighton
New_Geode	Pfnewgeode
Find_Geode	Pffindgeode
Number_Of_Geosets_Of	Pfgetnumgsets
not covered	*Pfnewscs*
not covered	*Pffog*
not covered	*Pfgetchanfrust*
not covered	*Pfgetchanbasefrust*
not covered	*Pfdcscoord*
not covered	*Pfdcsmat*
not covered	*Pfaddgset*
not covered	*Pfremovegset*
not covered	*Pfinsertgset*
not covered	*Pfreplacegset*
not covered	*Pfgetgset*

C.5. Vis_Sim-Stats (SGI Counterpart: libpfstats)

Low-Level-Services Routine	**SGI Routine**
Stats_Class_Of	Pfstatsclass
Chan_FStats_Of	Pfgetchanfstats
Set_Channel_Stats_Mode	Pfchanstatsmode
Draw_FStats	Pfdrawfstats

C.6. Utilities (SGI Counterpart: libpfutil)

Low-Level-Services Routine	SGI Routine
Initialize	Pfuinitutil
Cleanup	Pfuexitutil
Initialize_Input	Pfuinitinput
Cleanup_Input	Pfuexitinput
Get_Mouse	Pfugetmouse
Get_Events	Pfugetevents
Collect_Input	Pfucollectinput
Map_Mouse_To_Channel	Pfumapmousetochan
Collide_Setup	Pfucollidesetup
Ground_Collision	Pfucollidegrnd
New_Shared_Texture	PfuNewSharedTex
Get_Shared_Texture_List	PfuGetSharedTexList
Make_Texture_List	PfuMakeTexList
Download_Texture_List	PfuDownloadTexList
Get_Texture_Size	PfuGetTexSize

C.7. Windows (SGI Counterpart: gl)

Low-Level-Services Routine	SGI Routine
Foreground	Foreground
Set_Position	Prefposition
Allow_Resizing	Winconstraints
Open	Winopen

Bibliography

[Ada95] "AdaIC News," Ada Joint Program Office, Arlington, Virginia; Spring 1995.

[Bel95] Bell, Gavin, "OpenInventor Frequently Asked Questions (FAQ)," Silicon Graphics, Inc., Mountain View, California; 1995.

[Boo91] Booch, Grady, Object Oriented Design with Applications, Benjamin/ Cummings, Redwood City, California; 1991.

[Cer93] Cernosek, Gary, "ROMAN-9X: A Technique for Representing Object Models in Ada 9X Notation," Tri-Ada '93 Conference Proceedings, Association for Computing Machinery, Inc., Seattle, WA; September 1993.

[Cur95] Curry, Damon, Electronic Mail Message, Paradigm Simulation, Inc., Dallas, Texas; August 1995.

[Dia94] Diaz, Milton, "The Photo Realistic AFIT Virtual Cockpit," MS Thesis AFIT/GCS/ENG/94D-11, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; November 1994.

[For94] Fortner, Jonathan, "Distributed Interactive Simulation Virtual Cassette Recorder: A Datalogger with Variable Speed Replay," MS Thesis AFIT/GE/ENG/94D-10, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Gar93] Garlan and Shaw, "An Introduction to Software Architecture," Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Co., Singapore; 1993.

[Har94] Hartman and Creek, IRIS Performer Programming Guide, Silicon Graphics, Inc., Mountain View, California; 1994.

[How91] Howard, Hewitt, Hubbold, Wyrwas, A Practical Introduction to PHIGS and PHIGS Plus, Addison-Wesley Publishing Co., Workingham, England; 1991.

[IDD94] Interface Design Document for Clip, version 1.4, Loral Advanced Distributed Simulation, Bellevue, Washington; September 1994.

[Jan94] Janett, Hayes, and Miller, "CLIP: An Open Interface for Interactive Real-Time Visual Simulations," Loral Advanced Distributed Simulation, Bellevue, Washington; June 1994.

[Jan95] Janett, Annette, Electronic Mail Message, Loral Advanced Distributed Simulation, Bellevue, Washington; June 1995.

[Kay94] Kayloe, Jordan, "Easy_Sim: A Visual Simulation System Software Architecture with an Ada 9X Application Framework," MS Thesis AFIT/GCS/ENG/94D-11, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Loc94] Locke, John, "An Introduction to the Internet Networking Environment and SIMNET/DIS," Computer Science Department, Naval Postgraduate School, Monterey, California; January 1994.

[McL92] McLendon, Patricia, Graphics Library Programming Guide, Volume I, Silicon Graphics, Inc., Mountain View, California; 1992.

[Pro94] Prosise, Jeff, "Advanced 3-D Graphics for Windows NT 3.5: Introducing the OpenGL Interface, Part 1," Microsoft Systems Journal, M&T Publishing Co.; October 1994.

[Rei90] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Program Development Environment," IEEE Software; July 1990.

[RM95] Ada 95 Reference Manual: Language and Standard Libraries, Intermetrics, Inc., Cambridge, Massachusetts; 1995.

[Roh94] Rohrer, J.J., "Design and Implementation of Tools to Increase User Control and Knowledge Elicitation in a Virtual Battlespace," MS Thesis AFIT/GCS/ENG/94D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Rum91] Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Inc., New York, New York; 1991.

[Sch94] Schaffer, Allan, "SGI Performer Frequently Asked Questions (FAQ)," Silicon Graphics, Inc., Mountain View, California; 1994.

[Seg93] Segal and Akeley, "The OpenGL Graphics Interface," Silicon Graphics, Inc., Mountain View, California; 1993.

[She92] Sheasby, Steven, "Management of Simnet and DIS Entities in Synthetic Environments," MS Thesis AFIT/GCS/ENG/92D-16, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1992.

[Sny93] Snyder, Mark, "ObjectSim: A Reusable Object-Oriented DIS Visual Simulation," MS Thesis AFIT/GCS/ENG/93D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[Tal93] "Leveraging Object-Oriented Frameworks," Taligent, Inc., Cupertino, California; 1993.

[Tal94] "Building Object-Oriented Frameworks," Taligent, Inc., Cupertino, California; 1994.

[Tan88] Tanenbaum, Andrew, Computer Networks, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey; 1988.

[Van94] Vanderburgh, John, "Space Modeler: An Expanded, Distributed, Virtual Environment for Space Visualization," MS Thesis AFIT/GCS/ENG/94D-23, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, Ohio; December 1994.

[VPG94] Vega Programmer's Guide, Paradigm Simulation, Inc., Dallas, Texas; 1994.

Vita

Captain Shawn Michael Hannan was born on February 20, 1969 in Baltimore. He was raised in Harford County, Maryland, and graduated from Fallston High School in 1986. Hannan attended the University of Texas at Austin, from which he received a Bachelor of Arts in Computer Science in 1990. Captain Hannan was commissioned in the United States Air Force via the Reserve Officers' Training Corps, and began serving on active duty in the spring of 1991. His first assignment was as an Ada instructor with the 333d Training Squadron at Keesler Air Force Base, Mississippi. From Keesler, Hannan went to the Air Force Institute of Technology in Dayton, Ohio to pursue a Master's of Science in Computer Systems. His subsequent assignment was to the headquarters of the Air Force Command, Control, Communications and Computer Agency at Scott AFB, Illinois.

Permanent Address:
2800 Brockway Place
Kingsville, MD 21087

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1995	Technical Report; Thesis

4. TITLE AND SUBTITLE
OBJECTSIM 3.0: A SOFTWARE ARCHITECTURE FOR THE DEVELOPMENT OF PORTABLE VISUAL SIMULATION APPLICATIONS

5. FUNDING NUMBERS

6. AUTHOR(S)
Captain Shawn M. Hannan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/95D-05

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office ATTN: Mr. Gary Shupe
Center for Software
Joint Interoperability and Engineering Organization
5600 Columbia Pike, Suite 364
Falls Church, VA 22041

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A visual simulation software architecture is a reusable design for visual simulation applications. This thesis effort was the third stage in an ongoing refinement of such an architecture, named ObjectSim. The goals of this stage were to improve the architecture by eliminating its dependence on two platform-specific graphics libraries (named GL and Performer, from Silicon Graphics, Inc.), and to examine the potential for expanding the architecture to accommodate distributed simulations.

The effort resulted in a new version of the architecture which allows the development of visual simulation applications which take full advantage of the aforementioned libraries without calling those libraries directly. This capability substantially improves the potential portability of future applications.

ObjectSim also has other enhancements not found in its predecessors, but still does not accommodate distributed simulations. Insights into addressing the distributed simulation issue are, however, included in this thesis.

14. SUBJECT TERMS
Software Architecture, Visual Simulation

15. NUMBER OF PAGES
157

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL